Implementing Method of Moments on a GPGPU using Nvidia CUDA

A Thesis

Presented to

The Academic Faculty

By

Bikram Virk

In Partial Fulfillment

Of the Requirements for the Degree of

Master of Science in Electrical Engineering

Georgia Institute of Technology

Atlanta, GA

May, 2010

Implementing Method of Moments on a GPGPU using Nvidia CUDA

Approved by:

Prof. Andrew F. Peterson, Advisor
Electrical and Computer Engineering
Georgia Institute of Technology

Prof. Glenn S. Smith
Electrical and Computer Engineering
Georgia Institute of Technology

Prof. Emmanouil M. Tentzeris
Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: April 2nd, 2010

AKNOWLEDGEMENTS

I would like to thank my advisor Dr. Andrew F. Peterson for his invaluable help and guidance on this research topic. His constant support, feedback and knowledge have played a crucial role in achieving the results in this work. It has been a pleasure to have worked under him and the opportunities that were presented to me throughout this amazing journey.

I would also like to acknowledge the honor it brings to have Dr. Emmanouil M. Tentzeris and Dr. Glenn S. Smith on the committee to review the work and provide important feedback, as well as raising its value by approving it to their high standards.

Lastly, I would like to thank my parents Mr. Surinder P. Singh Virk and Harleen Virk for encouraging me to pursue all my goals, no matter how hard they might seem and the unconditional love and blessings that help motivate me to become a better person every day.

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

This thesis concentrates on the algorithmic aspects of Method of Moments (MoM) and Locally Corrected Nyström (LCN) numerical methods in electromagnetics. The data dependency in each step of the algorithm is analyzed to implement a parallel version that can harness the powerful processing power of a General Purpose Graphics Processing Unit (GPGPU). The GPGPU programming model provided by NVIDIA's Compute Unified Device Architecture (CUDA) is described to learn the software tools at hand enabling us to implement C code on the GPGPU. Various optimizations such as the partial update at every iteration, inter-block synchronization and using shared memory enable us to achieve an overall speedup of approximately 10. The study also brings out the strengths and weaknesses in implementing different methods such as Crout's LU decomposition and triangular matrix inversion on a GPGPU architecture. The results suggest future directions of study in different algorithms and their effectiveness on a parallel processor environment. The performance data collected show how different features of the GPGPU architecture can be enhanced to yield higher speedup.

1. INTRODUCTION

Maxwell's equations define the basics of any electromagnetic field problem. Simple situations such as a wavefront interacting with an aperture or the energy radiated by a dipole antenna can be solved analytically, at least to an approximate extent. But problems with complex geometries, such as an analog filter made from metal strips surrounded by dielectric material, cannot be solved by hand. For those problems, Maxwell's equations, or an equivalent equation such as the Electric Field Integral Equation (EFIE), can be solved using numerical techniques. A data set describing the problem geometry and materials can be used as an input to an algorithm specifically designed to solve electromagnetic problems for wave propagation, wave scattering, analog device modeling, etc. These numerical methods include the Finite Difference Technique for solving Partial Differential Equations (PDEs) and the Method of Moments (MoM) or Locally-corrected Nyström (LCN) techniques for solving integral equations.

As one would expect, the numerical solution of a complicated electromagnetic field problem can result in a large quantity of data to manipulate and result in excessive computation times. This thesis describes the details of the MoM methodology and analyzes its implementation, with particular emphasis on its compute time. As will be shown, problem sizes can easily grow large enough to require hours of computation.

The overall goal of the work described here is to determine ways to speed up the algorithm using new techniques. One way to get faster computation time is to execute the problem's solution in a parallel fashion, using a large number of computer processors. In this study, a basic MoM/LCN procedure is implemented in parallel using a General Purpose Graphics Processing Unit (GPGPU).

A GPGPU is a specialized processor that was developed to quickly perform a large number of calculations arising from screen or frame rendering purposes — in other words, to render a high resolution video or graphics image in real time on a computer screen. The GPGPU, or GPU for short, contains a large number of processors and associated local memory, but is restricted to a limited class of specialized instructions. Recently, the scientific community realized the parallel processing potential of a GPU for other applications, and created a demand for easier techniques to enable other types of

computations to be performed by a GPU. NVIDIA's Compute Unified Device Architecture (CUDA) was introduced as an interface to enable general-purpose programs written in the C language to execute with slight modifications on the GPGPU. More specifically, CUDA is a software layer or an Application Programming Interface (API) that enables a conventional central processing unit (CPU) to transfer instructions to a GPU.

In this investigation, we alter the traditional MoM/LCN algorithms to enable their execution in a parallel fashion on the GPU. In the following chapter we review these numerical methods for solving a simple type of electromagnetic scattering problem. Both techniques produce a matrix equation that must be created and solved, and those tasks define the computations to be performed. The subsequent chapter examines these computations in a step by step manner in order to determine the extent to which they can be performed in parallel. We also review the fundamentals of the CUDA API to give the reader an idea of the tools accessible with this software library. Finally, we implement the procedures in both serial (CPU) and parallel (GPU) versions in order to compare their actual performance for a range of problem sizes.

2. BACKGROUND

2.1 Theory of an Electromagnetic Scattering Problem

In this report we analyze solving electromagnetic problems that require large amounts of computation. This section provides the theory behind numerical formulations for these electromagnetic wave scattering problems.

Electromagnetic scattering involves the interaction of an incident electromagnetic wave with an object that might be made of a conducting material or some other material. Figure 1 shows the problem layout. In this investigation we limit our consideration to conducting objects. A typical example of a scattering problem is that encountered in radar applications: launch a wave at an approaching aircraft and study the reflected wave to determine parameters such as the position and velocity of the aircraft. The boundary value problem to be solved is therefore to determine the reflected wave for some specific target geometry. This requires the solution of Maxwell's equations with appropriate boundary conditions for that geometry. In this investigation, we consider an equivalent description of the problem in terms of an integral equation.

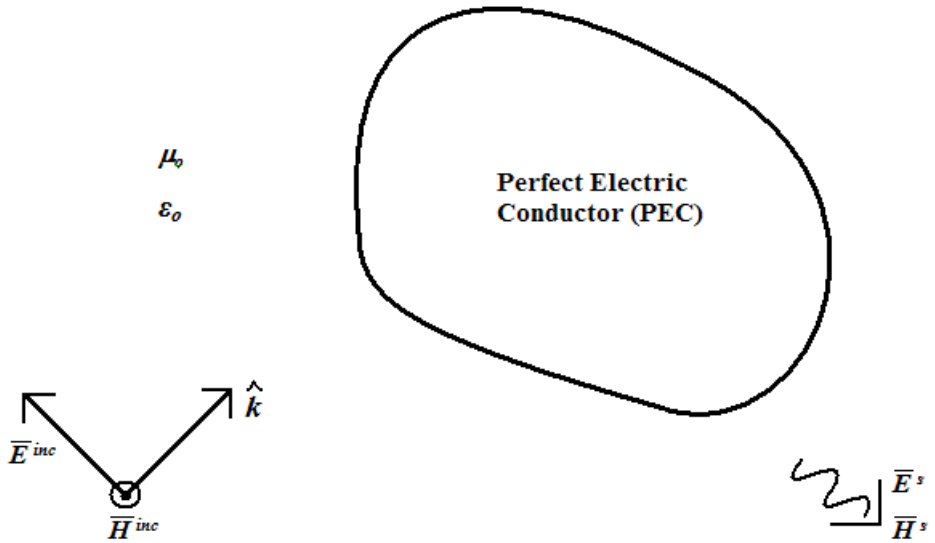

Figure 1 An inhomogeneity illuminated by an incident electromagnetic field.

In Figure 1, a conducting scatterer is illuminated by a field produced by a primary source located somewhere outside the scatterer. The fields produced by the primary source in the absence of the scatterer are denoted by the *incident fields* E^{inc} and H^{inc}. The fields induced on the scatterer act as equivalent secondary currents that, radiating in empty space, produce the *scattered fields* E^s and H^s. The total fields (which exist in the presence of the target) are the superposition of the incident fields and the scattered fields. For simplicity, we limit our consideration to two-dimensional problems, where all quantities are only functions of x and y.

An integral equation can be formulated by (1) imposing the condition that the incident field and the scattered field add up to the total field, and (2) by imposing a boundary condition. If the object is considered to be a perfect electric conductor, the appropriate boundary condition might be that the tangential electric field on the surface of the object is zero. This leads to the equation

$$\hat{n} \times \bar{E}^{inc} + \hat{n} \times \bar{E}^{s} = 0 \tag{1}$$

or equivalently

$$\hat{n} \times \bar{E}^{inc} = -\hat{n} \times \left\{ \frac{\nabla \nabla \cdot \bar{A} + k^2 \bar{A}}{j\omega\varepsilon_0} \right\}_S \tag{2}$$

where the equality holds on the surface of the conducting object. Equation 2 is known as the Electric Field Integral Equation (EFIE). The quantity $\bar{A}$ in Equation 2 is the magnetic vector potential, which is a convolution of the current density and the Green's function (to be defined below). For a normally incident Transverse Magnetic (TM) wave, the only non-zero component of the electric field is E_z, and the only current component present is J_z. In that case, the equation simplifies to

$$E_z^{inc} = jk\eta \int J_z(t')G(t,t')dt' \tag{3}$$

where the function G is known as the Green's function and has the form

$$G(t,t') = \frac{1}{4j} H_0^{(2)}(kR) \tag{4}$$

and where

$$R = \sqrt{[x(t) - x(t')]^2 + [y(t) - y(t')]^2} \tag{5}$$

Here, t is a parameter denoting the position on the contour of the cylinder.

The solution of equation (3) for J_z is generally too difficult to accomplish exactly. Instead, we seek a numerical solution. There are two different approaches that can be used, the method of moments and the locally-corrected Nystrom method. These are described below.

2.2 Method of Moments

We can obtain an approximate solution for J_z by dividing the cylinder boundary into cells as shown in Figure 2. The current induced over the cylinder can be approximated by pulse basis functions located at the center of each cell, defined by

$$p_n(t) = \begin{array}{ll} 1 & if\ t \in cell\, n \\ 0 & otherwise \end{array} \tag{6}$$

This implies that

$$J_z(t) \cong \sum_{n=1}^{N} j_n p_n(t) \tag{7}$$

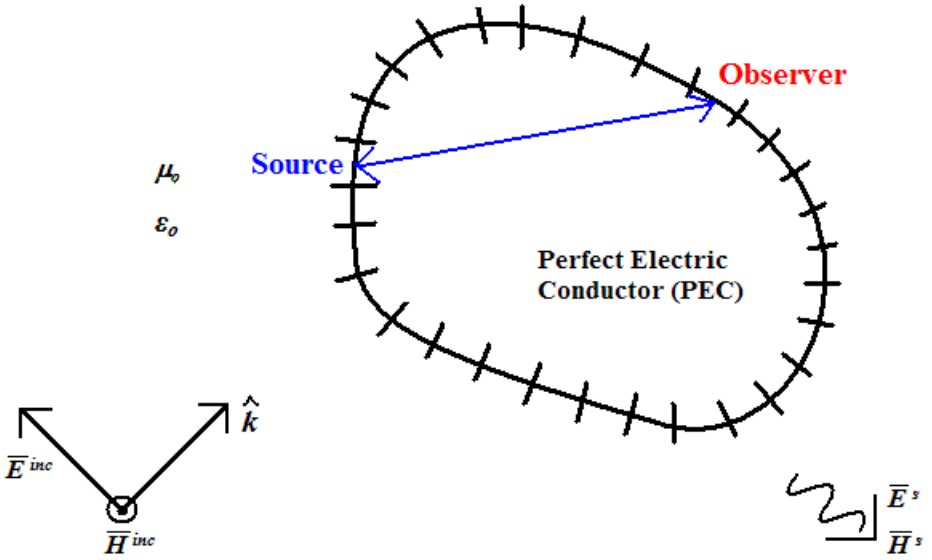

Figure 2 The object's surface divided into cells for approximation.

Combining Equations 7 and 3 we obtain

$$E_z^{inc}(t) = jk\eta \sum_{n=1}^{N} j_n \int_{\Gamma} p_n(t') \frac{1}{4j} H_0^{(2)}(kR) dt' \tag{8}$$

With this equation, the original problem of finding $J_z(t)$ can now be represented by the task of finding N unknown coefficients $\{ j_n \}$. To accomplish this task, we need N linearly independent equations. These equations may be obtained by enforcing Equation 7 at the centers of each of the N cells, to produce the N x N system

$$\begin{pmatrix} E_z^{inc}(t_1) \\ E_z^{inc}(t_2) \\ \vdots \\ E_z^{inc}(t_N) \end{pmatrix} = \begin{pmatrix} Z_{11} Z_{12} \cdots Z_{1N} \\ Z_{21} Z_{22} \cdots Z_{2N} \\ \vdots \\ Z_{N1} Z_{N2} \cdots Z_{NN} \end{pmatrix} \begin{pmatrix} j_1 \\ j_2 \\ \vdots \\ j_N \end{pmatrix} \tag{9}$$

Equation 9 now is the *discretized* or matrix form of equation 3. The approach outlined above is a simple example of a general procedure known as the *Method of Moments (MoM)*. The N x N matrix in the equation above is often called the *moment method impedance matrix* as it consists of mutual impedance between different cells in the model. In the integral form the elements of this matrix are given by

$$Z_{mn} = \frac{k\eta}{4} \int_{cell\,n} H_0^{(2)}(kR_m)\, dt' \tag{10}$$

where

$$R = \sqrt{[x_m - x(t')]^2 + [y_m - y(t')]^2} \tag{11}$$

and where (x_m, y_m) represents the phase center of the mth strip in the model. Equation 10 cannot be evaluated exactly, but if the cells are small compared to the wavelength, an approximation can be made of the form [1]

$$Z_{mn} \cong \frac{k\eta}{4} w_n H_0^{(2)}(kR_{mn}) \quad m \neq n \tag{12}$$

where

$$R = \sqrt{[x_m - x_n]^2 + [y_m - y_n]^2} \tag{13}$$

and where w_n denotes the transverse dimension of the n-th cell. As the Hankel function is infinite for $R_{mn} = 0$, the diagonal elements of the impedance matrix cannot be evaluated using Equation 12. It is important to evaluate the diagonal elements accurately since they significantly contribute to the solution. For small arguments, the Hankel function can be replaced by a power series expansion [7],

$$H_0^{(2)}(x) \approx 1 - \frac{x^2}{4} - j\,\frac{2}{\pi}\ln\frac{\gamma x}{2} + \frac{1}{2\pi} - \frac{1}{2\pi}\ln\frac{\gamma x}{2}\, x^2 + O(x^4) \tag{14}$$

where

$$\gamma = 1.781072418... \tag{15}$$

The leading terms of this expansion can be integrated to produce

$$\begin{aligned} \int_{cell\ n} H_0^{(2)}(kR_m)dt' &\cong 2\int_0^{w_m/2} 1 - j\frac{2}{\pi}\ln\left(\frac{\gamma k u}{2}\right) du \\ &= w_m - j\frac{2}{\pi}w_m\left(\ln\left(\frac{\gamma k w_m}{4}\right) - 1\right) \end{aligned} \tag{16}$$

Finally, the diagonal matrix entries are obtained as

$$Z_{mn} \cong \frac{k\eta w_m}{4}\left(1 - j\frac{2}{\pi}\left(\ln\left(\frac{\gamma k w_m}{4}\right) - 1\right)\right) \tag{17}$$

The solution of the matrix equation constructed from these expressions will produce an approximation for the equivalent current density J_z. In these problems, we assume the incident wave to be a uniform plane wave of the form

$$E_z^{inc}(x,y) = e^{-jk(x\cos\phi^{inc} + y\sin\phi^{inc})} \tag{18}$$

2.3 Locally Corrected Nystrom Method (LCN)

The Nyström method is an alternative to the MoM for obtaining numerical solutions of integral equations. In the following sections we introduce the Nyström and Locally Corrected Nyström (LCN) methods applied to perfectly conducting cylinders in two dimensions. These methods also divide the surface of the target under consideration into *N* cells, but differ from the MoM in the fact that they use a quadrature rule to define the representation of the integral over each cell.

The Nyström Method

In the Nyström method, the integral over each cell is replaced with a quadrature rule. The values of the unknown current density J_z at the nodes (sample points) of the rule are the unknown quantities to be determined. By enforcing the integral equation (as a sum of quadrature rules over all the cells) at each node point, we obtain a system of linear equations. If a q-point quadrature is used in each cell, there are Nq unknowns to be calculated. Note the difference in the representation of the current density of the Nyström and MoM approaches: In the MoM approach, the surface current J_z is represented by an expansion in basis functions, whose coefficients are the unknowns to be determined. For the Nyström method, the unknowns to be determined are the samples of the current density at the quadrature nodes. While this may seem to be a fundamentally different approach, it is really just a change in the basis functions used to represent J_z.

The classical Nyström method has the drawback that it cannot be used for integral equations with singular kernels, since the diagonal entries of the system matrix become infinite in that case. Unfortunately, most of the equations of interest exhibit singular kernels, including the EFIE. The Locally Corrected Nyström (LCN) method was developed to correct this difficulty. Before explaining the LCN method, we digress to review the use of quadrature.

Quadrature

A quadrature rule is a specific set of weights $\{ w_i \}$ and nodes $\{ u_i \}$ associated with the summation of the form

$$\int_0^1 f(u)\,du \cong \sum_{i=1}^{N} w_i f(u_i) \tag{19}$$

that approximates the integral. As seen in Equation 19, the integral is calculated with the sum of N terms. The algorithms mentioned herein use the *Gauss-Legendre* quadrature rule as it provides the most accuracy for the least amount of computation when compared to other rules like *Romberg* or *Newton-Cotes* integration [8]. The main reason for this is

that the Gauss-Legendre quadrature node locations are optimized to provide maximum accuracy. For example, consider the general cubic polynomial,

$$f(u) = a + bu + cu^2 + du^3 \tag{20}$$

As the degree of freedom is four, we can use two node values and two weights to obtain an integration rule. Such a rule has the form

$$\int_0^1 f(u)\,du = w_1 f(u_1) + w_1 f(1-u_1) \tag{21}$$

When Eq. 21 is applied to Eq. 20, we can obtain the weights and nodes by solving the following nonlinear equations:

$$\int_0^1 a\,du = a = w_1 a + w_1 a \tag{22}$$

$$\int_0^1 bu\,du = \frac{b}{2} = w_1 bu_1 + w_1 b(1-u_1) \tag{23}$$

$$\int_0^1 cu^2\,du = \frac{c}{3} = w_1 c(u_1)^2 + w_1 c(1-u_1)^2 \tag{24}$$

$$\int_0^1 du^3\,du = \frac{d}{4} = w_1 d(u_1)^3 + w_1 d(1-u_1)^3 \tag{25}$$

These equations produce the following solution:

$$w_1 = \frac{1}{2}, u_1 = \frac{1}{2} - \frac{1}{2\sqrt{3}} \tag{26}$$

$$w_2 = \frac{1}{2}, u_2 = 1 - u_1 = \frac{1}{2} + \frac{1}{2\sqrt{3}} \tag{27}$$

Equations 26 and 27 define the 2-point Gauss-Legendre rule. We can develop a 3-point rule for a polynomial of degree 5 and similarly a 4-point rule for degree 7. Each *p*-point rule consists of the *p* weights and *p* relative node locations for that rule. Now, having discussed the Gauss-Legendre quadrature rule we return to the LCN.

The Locally Corrected Nyström Method

This method was initially described by Gedney et al. [5] and Canino et al. [4] to enable the Nyström analysis of integral equations with singular kernels. The main idea is to synthesize a new kernel for cases when source and observer cells are in close proximity [4,5]. In situations when the kernel K is infinite, the Nyström approximation of integral operator over a cell

$$\int_n J_z(t')K(t_{mj},t')\,dt' \cong \sum_{i=1}^{q} w_{ni} J_z(t_{ni})K(t_{mj},t_{ni}) \tag{28}$$

fails due to the infinite value of K. In the LCN procedure, this integral is replaced by

$$\int_n J_z(t')K(t_{mj},t')\,dt' \cong \sum_{i=1}^{q} w_{ni} J_z(t_{ni})L(t_{mj},t_{ni}) \tag{29}$$

where $L(t_{mj}, t_{ni})$ is a new "corrected" kernel that is bounded and therefore enables the solution process to proceed. However, the corrected kernel L must be synthesized with care in order that the two summations in (28) and (29) produce equivalent results.

In Equation 29, the current J_z can be thought of as a source function that produces the field given by the integral on the left-hand side of the equation. If we select a set of source functions $\{B_k(t)\}$ and impose the condition that the sum on the right-hand side of Eq. 29 equals the integral on the left for each of those source functions, evaluated at the nodes of the quadrature rule used on the right-hand side, we obtain a system of equations

$$\begin{aligned} &\sum_{i=1}^{q} w_{ni} B_k(t_{ni}) L(t_{mj}, t_{ni}) \\ &= \sum_{i=1}^{q} B_k(t_{ni}) \left\{ w_{ni} L(t_{mj}, t_{ni}) \right\} \\ &= \int_{ncell} B_k(t') K(t_{mj}, t') dt' \end{aligned} \tag{30}$$

that can be solved to produce the necessary samples of the corrected kernel L. In other words, for every basis function B_k and every location t_{mj} inside the cell, we force the summation involving the corrected kernel L to produce the true near fields of basis function B_k (the integral on the left). If we use q basis functions for a q point quadrature rule, the condition in (30) leads to the following system of linear equations:

$$\begin{bmatrix} B_1(t_{n1}) B_1(t_{n2}) \cdots B_1(t_{nq}) \\ B_2(t_{n1}) B_2(t_{n2}) \cdots B_2(t_{nq}) \\ \vdots \\ B_q(t_{n1}) \cdots\cdots\cdots\cdots B_q(t_{nq}) \end{bmatrix} \begin{bmatrix} w_{n1} L(t_{mj}, t_{n1}) \\ w_{n2} L(t_{mj}, t_{n2}) \\ \vdots \\ w_{nq} L(t_{mj}, t_{nq}) \end{bmatrix} = \begin{bmatrix} \int_{ncell} B_1(t') K(t_{mj}, t') dt' \\ \int_{ncell} B_2(t') K(t_{mj}, t') dt' \\ \vdots \\ \int_{ncell} B_q(t') K(t_{mj}, t') dt' \end{bmatrix} \tag{31}$$

Eq. 31 can be solved for $L(t_{mj}, t_{n1})$ through $L(t_{mj}, t_{nq})$. After computing L from this linear system, the summation in Eq. 29 should provide accurate near fields for any current density $J_z(t)$ that can be approximated well by the basis functions. In the actual LCN algorithm discussed below, the impedance matrix entries are computed using the corrected kernel L whenever the source-observer cell distance falls below a pre-specified value. The diagonal matrix entries are one place where this rule is applied as the observer and source cells are near each other.

LCN applied to TM EFIE

The Electric Field Integral Equation (EFIE) for TM polarization is given by

$$E_z^{inc}(t) = jk\eta \int_{\Gamma} J_z(t') \frac{1}{4j} H_0^{(2)}(kR)\, dt' \tag{32}$$

where t and t' are parametric variables defined on the contour Γ of the surface, $H_0^{(2)}$ is the zero order Hankel function of the second kind, and

$$R = \sqrt{[x(t) - x(t')]^2 + [y(t) - y(t')]^2} \tag{33}$$

In the classical Nyström method, a q-point Gauss-Legendre quadrature rule is used within each cell to define the current density samples at each node. A system of order Nq is obtained such that

$$ZJ = E^i \tag{34}$$

where Z is the Nq by Nq matrix, J is a column vector of the unknown samples, and E^i is a column vector containing samples of the incident field at the node locations within each cell. The impedance matrix elements for source and observer locations with sufficient distance between them are calculated by

$$Z_{mj,ni} = w_i \frac{k\eta}{4} H_0^{(2)}(kR_{mj,ni}) \tag{35}$$

where m and n denote the observer and source cells, and j and i the observer and source nodes, respectively. The parameters $\{w_i\}$ are the weights from the quadrature rule.

For near observer-source combinations where the classical Nyström method can not be applied due to singularity in the kernel, the LCN is used to produce the "corrected" kernel. The entries in the impedance matrix are now calculated using

$$Z_{mj,ni} = w_i L(t_{mj}, t_{ni}) \tag{36}$$

where necessary values of L are obtained by the repeated solution of the system

$$\sum_{i=1}^{q} w_{ni} B_k(t_{ni}) L(t_{mj}, t_{ni})$$

$$= \frac{k\eta}{4} \int_{ncell} B_k(t') H_0^{(2)}(kR_{mj})\, dt' \qquad k = 1,2,...,q \qquad (37)$$

This completes the mechanics of the LCN approach for the TM EFIE.

Improving solution accuracy in MoM and LCN

In both the MoM and LCN procedures, we can increase the accuracy of the numerical solution by dividing the geometry into smaller cells. This is commonly known as *h-refinement*. Another approach is to use *p-refinement*, which involves an improved representation for the current. In MoM, this is achieved by using higher degree polynomials as basis functions. The evaluation of an element of the MoM impedance matrix is more difficult for higher degree polynomials, and must usually be done with high-accuracy adaptive numerical quadrature. Thus *p*-refinement with the MoM increases the amount of computation required per matrix entry. However, in the Nyström approach, *p*-refinement is obtained by using a quadrature rule with more points. That approach does not require more computation per matrix entry. Thus, for high accuracy, the computational cost of calculating the weighted residuals in the MoM approach is much more expensive than computing samples of the kernel in the Nyström approach [8]. Thus, the LCN approach may be a more efficient choice for high accuracy, high order computations.

Summary

In summary, the MoM and LCN approaches involves the steps of (1) creating a model of the desired object, (2) creating the N by N matrix, (3) solving the matrix equation to obtain the coefficients or samples of the current density, and (4) computing any other information (far fields, for instance) that the user might desire. Although there are some differences between the MoM and LCN procedures, those differences are relatively minor. For either technique, these tasks must be implemented by means of a

computer program. The present investigation is concerned with exploiting the architecture of a GPU to enhance the efficiency of these computations.

2.4 Graphics Processing Unit (GPU)

A Graphics Processing Unit (GPU) is a specialized processor that offloads 3D graphics rendering from the Central Processing Unit (CPU). The first few generations of GPUs contained a fixed function graphics pipeline whose main purpose was to create a visual image. This task typically involved an input data stream of input vertices that were to be shaded by parallel compute units in the GPU to produce a visual rendering. Modern GPUs have programmable shaders that are used to draw different aspects of a visual/graphics pipeline. This feature, and the high amount of parallelism in a GPU, make it very suitable for running certain algorithms more efficiently than on the traditional CPU. Thus, there is a natural desire to exploit the available GPUs to speed up scientific and engineering computations. There have been many works in the literature that describe efforts to use GPUs to speed up algorithms, including those for protein DNA structures, aerospace design, financial models, and many more. For example, in the area of electromagnetics, Peng et al. [6] have implemented the previously-described Method of Moments (MoM) analysis procedure on a GPU. They accomplished this by redefining the problem input in terms of vertices and textures, to exploit the available GPU instruction set. Using this method, they were able to achieve a factor of 30 speedup in filling the impedance matrix and an overall speedup of a factor of 20.

As an example, the TESLA GPU is Nvidia Corporation's first dedicated General Purpose GPU (GPGPU). It offers a peak performance of 4.140×10^{12} Single Precision (SP) Floating Point Operations per Second (4140 GigaFLOPS or GFLOPS) [10]. For comparison, Peng et. al. used a slower GPU with only 40 GFLOPS of performance. Nvidia plans on releasing a new GPGPU based on their new "fermi" architecture in early 2010 with eight times the performance of current GPUs [10]. In the next section, we discuss a programming tool provided by Nvidia to create kernels written in the traditional C programming language to run on the GPU. Below, we define a few terms that are used in the GPU architecture or High Performance Computing (HPC) domain:

FLOPS:

This term is an abbreviated version of **F**loating **P**oint **O**perations **P**er **S**econd. It is a good measure of the peak performance that any processor (either a CPU or a GPU) can offer for a numerical application with many floating point operations. The GPU card *Nvidia GTX280* used for implementation in this thesis offers a peak performance of 933 GFLOPS.

Graphics pipeline:

A graphics or rendering pipeline is a set of functions that transforms the input information about a scene into a visual frame that can be displayed on any monitor. These functions are known as *shaders*. In the early implementations of GPUs, the graphics pipeline consisted of fixed-function shaders that only allowed a few choices as to how to render a scene.

Programmable Shader:

In the computer graphics domain, a shader is a set of instructions that are performed on input data or vertices for calculating rendering effects. In a computer system, the CPU programs the GPU to perform these operations on input data from the CPU or memory. The rendering pipeline over the past few GPU generations has included more and more programmable aspects to each shading stage, which allows the programmers to produce more realistic affects for rendering specific game scenes. Thus there is a trend toward giving the programmer more control over how to use the hardware, and this is the reason that other researchers have been attracted to exploit the parallel nature of the GPU architecture to execute general purpose programs.

SIMD:

The parallelism in a GPU architecture exploits the fact that during rendering operations, a single instruction can be executed on multiple data sets when processed by the same shader stage. For example, let us assume that the GPU

is calculating the color on a surface due to light. In this step, the surface normals at each of the triangles that make up the surface are calculated and shaded as dark or light based on the direction of its normal with respect to the light source. A surface can be divided into hundreds or even thousands of triangles, so to obtain a good frame rate, or rendering speed, this must be done in a parallel fashion. Besides having multiple processing cores, there may be another level of parallelism within a processor's Arithmetic Logic Unit (ALU). This type of parallelism is known by the names *vector processing* or *array processing* and also by the acronym SIMD, which stands for **S**ingle **I**nstruction **M**ultiple **D**ata. As explained previously, the same shader instruction can be performed on multiple data sets. For example, a SIMD4 implementation might involve a total data set with 128 bit subsets packed next to each other in a 512 bit packet. The ALU then simultaneously performs the same operation on each subset this packet, yielding four unique results in one processor cycle. GPU architecture is heavily based on SIMD operations.

2.5 Compute Unified Device Architecture (CUDA)

Early GPU realizations were limited to specific instruction sets associated with image rendering. Users who wished to employ the GPU for other tasks were forced to translate their specific task into that limited instruction set. The instruction set also varied with the GPU manufacturer and generation. To enable users to employ GPUs for other computational tasks, Nvidia created CUDA. CUDA is a general purpose environment that allows programmers to control a GPU using the C programming language. Other languages such as FORTRAN, OpenCL, and DirectCompute are also supported [10]. A program written in CUDA should execute efficiently on any GPU model.

In CUDA, the CPU and GPU are known as the host and device, respectively. In a simple CUDA program 1) the main code is initially executed only on the host, 2) the host allocates memory on the device in order to pass input data to the device, 3) the initial data is copied from the host to the device memory, 4) the host initiates the kernel, which is a

routine that executes on the GPU device, with appropriate input parameters, 5) the kernel executes, modifying the data stored within the device memory, and finally 6) when the kernel finishes the host copies data back from the device memory to an output array in host memory.

Programming model

The CUDA programming model presents the GPU as a massively multi-threaded processor. At any given time there can be one kernel executing, but each kernel can be executing with thousands of threads in parallel. Compared to CPU threads, GPU threads have little creation overhead and have almost instant switching time on a processing element. CUDA achieves efficiency and speedup by executing thousands of threads in parallel where a multi-core CPU system can support only a few parallel threads.

Nvidia GPUs consist of multiple *Streaming Multiprocessors* (SMs), which in turn each contain multiple thread processors. As seen Fig 1.3 below, a thread execution manager schedules threads for each SM. For inter-thread communication, each SM contains a block of *Shared Memory*, with a size of 16KB for the Nvidia GTX280.

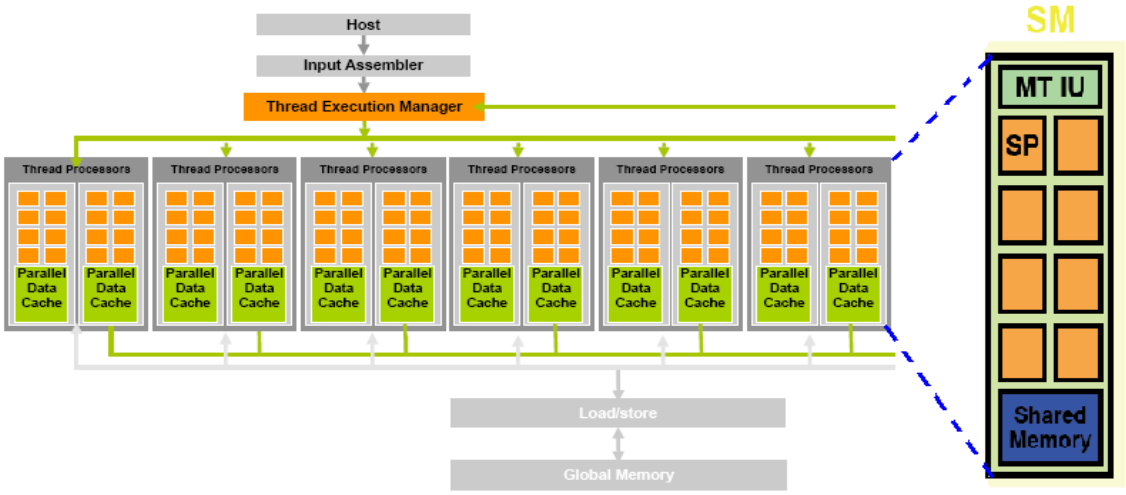

Figure 3 GPU hardware layout.

A kernel is executed as a part of a *grid* of thread *blocks*. A *block* is a set of threads that execute on one multiprocessor (SM). Threads inside a block can synchronize and share data through the shared memory inside each SM. Threads are scheduled in a set of 32 called a *warp* on each thread processor within the SM. This allows the programming model to mask the memory access latency by issuing the memory accesses for one warp and then scheduling another warp in its place while the replaced warp waits on its memory accesses to return. In the kernel, variables that identify the configuration like number of blocks, number of threads in a block, block ID and thread ID are provided. This allows every thread to access input and output data in a parallel fashion. These variables are:

- gridDim.[x|y|z] => Dimensions of a grid, or the number of blocks in x, y, and z
- blockDim.[x|y|z] => Dimensions of a block, or the number of threads in x, y, and z
- blockIdx.[x|y|z] => Block ID (coordinates) in the grid
- threadIdx.[x|y|z] => Thread ID (coordinates) in the block

The code example below shows the method in which different instances of a kernel execute different sets of data in parallel. We use the block ID, block Dim, threaded so that every kernel accesses different parts of the array in parallel:

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
a[idx] = a[idx] * a[idx];
```

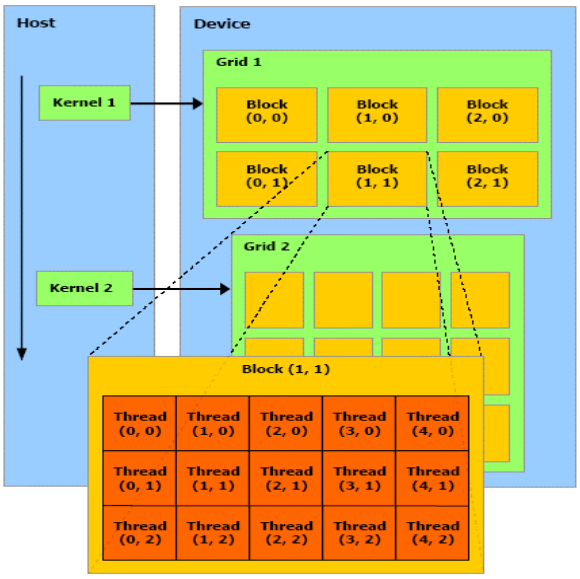

Figure 4 CUDA Programming Model.

The memory on a GPU is organized into many levels that are described below as well as shown in Figure 5 [11]:

Registers

- Fastest access time for a thread on a SM
- Allocated only during the execution of the thread

Shared Memory

- Potentially as fast as the register memory depending on access patterns
- Accessible by any thread of a block, hence used for inter-thread communication
- Allocated only during the lifetime of the block

Global Memory

- Much slower than register or shared memory accesses. For example, a global memory access can take 200-300 clock cycles.
- Accessible from host or device
- Allocated only during the lifetime of the main code or application

Local Memory

- Not physically defined as a separate entity or cache. Is a logically partitioned area of the global memory so accesses time is the same.
- Allocated only during the lifetime of a thread

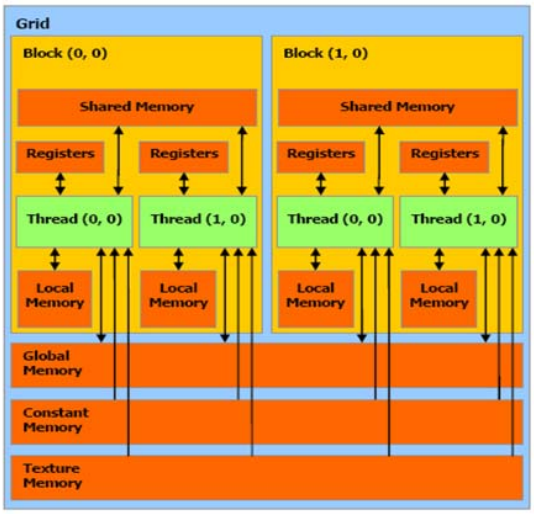

Figure 5 CUDA Memory Model.

CUDA program example

We now show a simple CUDA program that squares every element of an array. The steps in this process are:

1. In the host code, allocate input array on host and the device
2. Initialize the input array elements on the host
3. Copy initialized array data from the host to the device
4. Launch device kernel with *N* blocks and *K* threads per block.
5. After the kernel has completed, copy the result back from the device to the host.

The code below excludes details that are not related to the programming style [11]:

```
1.    __global__ void square_array(float *a, int N)
2.    {
3.      int idx = blockIdx.x * blockDim.x + threadIdx.x;
4.      if (idx<N) a[idx] = a[idx] * a[idx];
5.    }
```

```
6.    int main (void)
7.    {
8.      float *a_h, *a_d;
9.      const int N = 10;
10.     size_t size = N * sizeof(float);
11.     a_h = (float *)malloc(size);
12.     cudaMalloc((void **) &a_d, size);
13.     for (int i=0; i<N; i++)
14.         a_h[i] = (float)i;
15.     cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
16.     int block_size = 4;
17.     int n_blocks = N/block_size + (N%block_size == 0 ? 0 : 1);
18.     square_array <<< n_blocks, block_size >>> (a_d, N);
19.     cudaMemcpy(a_h, a_d, size, cudaMemcpyDeviceToHost);
20.     free(a_h); cudaFree(a_d);
21.   }
```

In the code above, execution starts in the host machine at line 6 with the main function call. First, we declare pointers for arrays *a_h,a_d* as the host and device arrays respectively. In line 10, we declare the size of the each array, which would be the number of elements *N* (10) multiplied by the size of each float. Line 11 allocates the memory for the host array (*a_h*). In the next line we allocate the array (*a_d*) on the device memory. In lines 13 and 14 a loop is implemented to initialize every value in the *a_h* array. Line 15 copies the initialized host array to the device array. Line 16 defines the number of threads per block. From this we calculate the number of blocks in line 17. The number of blocks is assigned such that the total number of threads is equal to the number of elements. In line 18 the device kernel is called with the configuration defined between '<<<' and '>>>'. Next, the control goes to line 1 or the device kernel that is defined by giving it a '__global__' identifier. In line 3, we use the id of the block (*blockIdx.x*), the number of threads in the block (*blockDim.x*), and the thread id (*threadIdx.x*). Once this unique id (*idx*) is calculated, the data is calculated for this index for this thread.

It is important to understand the difference in calling a function on CPU and on a GPU. For a CPU, in general a function call consists of only one instance of that kernel or set of instructions processing the data, whereas in a GPU when a kernel is called, the instructions are issued in parallel on steaming multiprocessors. Once the calculations are made we copy the result back from the device to the host at line 19. To finish, we de-allocate the arrays in line 20.

3. METHODS

In this chapter we examine the implementation of the Method of Moments (MoM) and Locally Corrected Nyström (LCN) approaches. Since these involve similar procedures of creating and solving a system of linear equations, the primary implementation steps are common to both. First, we present the single threaded version of the algorithms and identify the inefficiencies and dependencies that need to be considered to implement them on the GPU using CUDA. Once examined, we present the implementation details of the GPU versions of each step of the algorithms.

3.1 CPU implementation

The MoM and LCN algorithms require two major tasks, 1) creating the system of equations, and 2) solving the system of equations. The first step of task 1 is to read in the problem geometry, consisting of the (x,y) coordinates of nodes on the two-dimensional scatterer surface and some information about their connectivity. These coordinates and connectivity are used to define the precise location of each cell on the target surface. Using the cell locations, the impedance matrix, *Zmn* is constructed using the expressions in Chapter 2. In task 1, we also calculate the incident electric field intensity at the required locations. Task 1 produces the *Zmn* matrix and the *Ez* vector in the equation

$$Zmn \;.\; Ji \;=\; Ez \tag{38}$$

Task 2 is to solve this linear system of equations to determine *Ji*. Our approach is to use the Crout LU decomposition algorithm to convert *Zmn* into a lower and upper triangular form. Then we find the inverse of these triangular matrices and multiply them with *Ez* to obtain *Ji*. A summary of these tasks is detailed below, including pseudo-code implementations.

3.1.1 Task 1: Creating the system of equations

The first task involves loading the input data, consisting of the (x,y)-coordinates of the cells and nodes that describe the target geometry, along with some connectivity

information, and the direction and strength of the incident electric field. The output from this step is the complete *Nq* by *Nq* complex-valued *Zmn* matrix, which is generated using equations 12 and 17 (for the MoM) or equations 35 and 36 (for the LCN). In addition, this task involves the computation of the *Nq* by 1 complex-valued *Ez* column vector, which is determined from

$$E_z^{inc}(x,y) = e^{-jk(x\cos\phi^{inc} + y\sin\phi^{inc})} \tag{39}$$

Pseudo-code for *Zmn* matrix computation:

```
1.    Read in (x,y) coordinates for each node from input file into
      X(N), Y(N).
2.    for observer 0→N-1
3.          for source 0→N-1
4.                R = Distance between the observer and source cell
5.                if R < LCN_distance
6.    Apply LCN quadrature rule to calculate the pxp sub-matrix for
      this source-observer combination.
7.    else
8.    Apply MoM quadrature rule to calculate the pxp sub-matrix for
      Zmn(observer,source)
9.    if end
10.   end for
11.   end for
```

There is no dependency between any elements of the *Zmn* matrix. It should be noted that for a geometry model with *N* cells, using *p* unknowns within the MoM or a *p*-point quadrature rule per cell with LCN produces a system of order *Np* containing $(Np)^2$ independent entries. Figure 6 illustrates this operation for an observer cell *A* and a source cell *B*, each with $p = 3$ nodes (corresponding, for intance, to 3 LCN quadrature nodes per cell). The *p*x*p* submatrix of *Zmn* that arises from these interactions is shown.

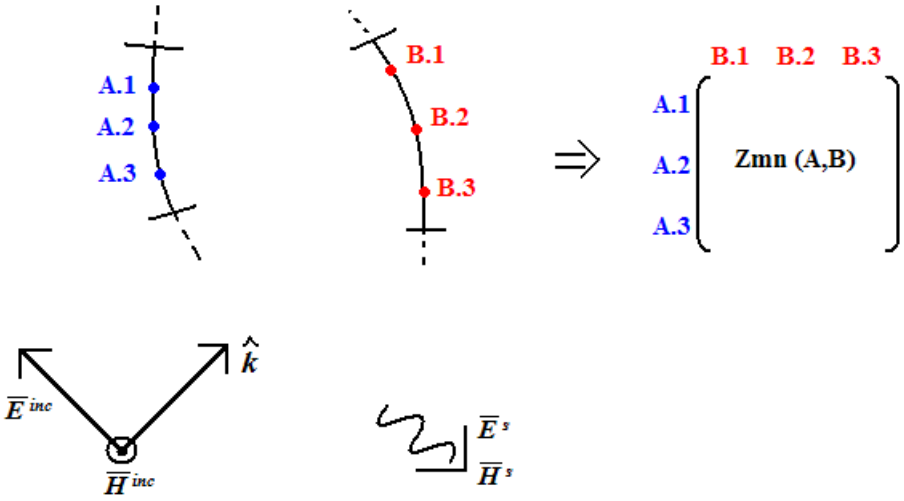

Figure 6 Use of 3-point quadrature in calculating *Zmn*.

Pseudo-code for the *Ez* column vector computation:

```
1.    Read in (x,y) coordinates for each node from input file into
      X(N), Y(N) and with incident angle ϕ.
2.    Allocate Ez vector or array in host.
3.    for i 0→N-1
4.          Ez(i) = exp(-jk * ( X(i)cos(ϕ) + Y(i)sin(ϕ)));
5.    end for
```

There is no dependency between the calculations of these *Ez* values. Hence, each element can be calculated in a parallel fashion.

3.1.2 Task 2: Solution of the linear system of equations

After task 1 is completed, the impedance matrix *Zmn* of order *Np* and the excitation column vector *Ez* of size *Np* have been computed and stored in memory. The solution of this linear system can be further divided into three steps: a) Use Crout's algorithm to decompose the *Zmn* matrix into a lower and upper triangular matrix *LU*, b)

Find the inverse of the *L* and *U* matrices and store them in *invLU*, and finally c) multiply the excitation vector by the inverted lower and upper matrices in *invLU* to obtain the solution vector.

3.1.2.1 Decomposing Zmn into an Upper and Lower matrix (LU)

In this study, we choose to solve the system of equations by first decomposing the *Zmn* matrix into lower and upper triangular form using Crout's algorithm. The upper and lower triangular matrices are stored in place of the original matrix, where all the diagonal elements of the lower triangular matrix are assumed to have a value of one. The equation below describes this method of storing the lower and upper matrices in one:

$$\begin{matrix} 1 & 0 & 0 & 0 & 0 \\ L_{21} & 1 & 0 & 0 & 0 \\ L_{31} & L_{32} & 1 & 0 & 0 \\ L_{41} & L_{42} & L_{43} & 1 & 0 \\ L_{51} & L_{52} & L_{53} & L_{54} & 1 \end{matrix} + \begin{matrix} U_{11} & U_{12} & U_{13} & U_{14} & U_{15} \\ 0 & U_{22} & U_{23} & U_{24} & U_{25} \\ 0 & 0 & U_{33} & U_{34} & U_{35} \\ 0 & 0 & 0 & U_{44} & U_{45} \\ 0 & 0 & 0 & 0 & U_{55} \end{matrix} = \begin{matrix} U_{11} & U_{12} & U_{13} & U_{14} & U_{15} \\ L_{21} & U_{22} & U_{23} & U_{24} & U_{25} \\ L_{31} & L_{32} & U_{33} & U_{34} & U_{35} \\ L_{41} & L_{42} & L_{43} & U_{44} & U_{45} \\ L_{51} & L_{52} & L_{53} & L_{54} & U_{55} \end{matrix} \quad (40)$$

(Note that the "+" sign in Equation 40 does not refer to conventional addition, but denotes the combination of the two matrices without the diagonal values of the first.) Below we present the pseudo-code implementation of the Crout algorithm:

Pseudo-code for the Crout decomposition algorithm:

```
# INITIALIZATION
1.    For i 0→N-1
2.          For j 0→i-1
3.                LU(i,j) = LU(i,j) / LU(i,i)
4.          end for
5.    end for
# ALGORITHM
1.    For j 1→N-1
2.          For i 0→j              # LOWER TRIANGLE
3.                For k 0→i-1
```

```
4.                      SUM = SUM + LU(i,k) * LU(k,j)
5.                end for
6.                LU(i,j) = Zmn(i,j) – SUM;
7.          end for
8.          For i j+1→N-1          # UPPER TRIANGLE
9.                For k 0→j-1
10.                     SUM = SUM + LU(i,k) * LU(k,j);
11.               end for
12.               LU(i,j) = ( Zmn(i,j) - SUM ) / LU(j,j);
13.         end for
```

In the Crout algorithm, every element has a dependency on the elements above and to the left of that element. In other words,

$$LU(i,j) \leftarrow LU(i, 0 \rightarrow i) \ \& \ LU(0 \rightarrow i, j) \qquad \text{when } i<j$$

Figure 7 shows this dependency for the LU(i,j) element. In this case, if i=10 and j=5, the element is computed from

$$LU(10,5) = LU(10,0)*LU(0,5) + LU(10,1)*LU(1,5) + LU(10,2)*LU(2,5) + LU(10,3)*LU(3,5) + LU(10,4)*LU(4,5)$$

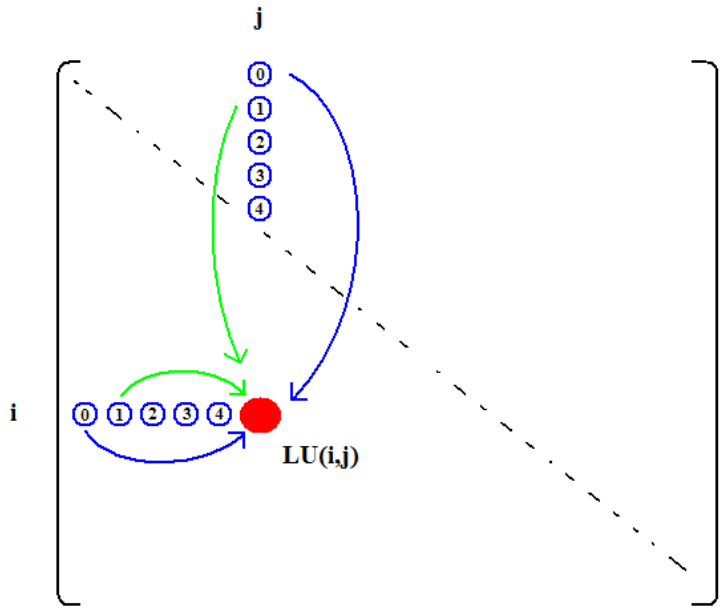

Figure 7 Crout's algorithm dependency.

3.1.2.2 Finding the inverse (invLU) of the Upper and Lower Triangular matrices

In the previous step, the impedance matrix *Zmn* was decomposed into lower and upper triangular matrices. The matrix that stores these values is known as *LU*. The equation to be solved now has the form

$$Zmn \;.\; Ji \;=\; Ez$$

$$\Rightarrow \; LU \;.\; Ji \;=\; Ez \qquad (41)$$

The triangular matrices *L* and *U* are stored in a single square matrix, whose diagonal elements are affiliated with *U* (the diagonal elements of *L* have a value of 1 and do not need to be stored). The pseudo-code for the algorithm used to invert these two triangular matrices is shown below, the results of which are stored in a matrix *invLU*. In this algorithm, *invLU* is not the inverse of the *LU* matrix as a whole but instead is a single square matrix used to store the L^{-1} and U^{-1} matrices for convenience, in the same way that the *LU* matrix stores the *L* and *U* matrices.

Pseudo-code for the inverse of the triangular factors:

```
1.    For i 0→N-1                          # INITIALIZATION
2.          For j 0→N-1
3.                invLU(i,j) = 0
4.          end for
5.    end for
6.    For j 0→N-2                          # INVERSE OF LOWER TRIANGLE
7.          For i j+1→N-1
8.                ITMP = 0
9.                For m j→i-1
10.                     ITMP = ITMP + LU(i,m)*invLU(m,j);
11.               end for
12.               invLU(i,j) = (-1/LU(i,i)) * ITMP;
13.         end for
14.   end for
15.   For i 0→N-2                          # INVERSE OF UPPER TRIANGLE
16.         For j i+1→N-1
17.               ITMP = 0
18.               For m i→j-1
19.                     ITMP = ITMP + invLU(i,m)*LU(m,j);
20.               end for
21.               invLU(i,j) = (-1/LU(j,j)) * ITMP;
22.         end for
23.   end for
```

The above algorithm can be divided into three parts: initialization, finding the inverse of the lower triangle and finding the inverse of the upper triangle. The initialization section has no dependency among its iterations. So, given a parallel computing system with P independent execution modules, the speed-up of this section can theoretically be a factor of P. The actual speedup would be less than P due to overhead associated with copying input data to different modules for calculation and then copying the results back into one location.

For calculating the inverse of the lower triangular matrix, there is no dependency across matrix columns. But, elements within any column have to be calculated in a serial fashion as every element is dependent on the values of the elements above it. Similarly, for calculating the inverse of the upper triangular matrix, the calculations have to be done in series per row. There is no dependency among the rows and they can be calculated in parallel.

3.1.3.3 Multiplying the inverse upper and lower matrices with the column vector Ez

In this step, the inverse values of the lower and upper triangular representations of the input impedance matrix *Zmn* have been computed. These matrices are stored in *invLU*, where the diagonal elements of L^{-1} have a value of 1, so that the diagonal elements of U^{1} can be stored in *invLU*. From Eq. 42, this part of the algorithm involves multiplying the inverse of *L* with *Ez* and then multiplying the result with the inverse of the upper triangular matrix.

$$\begin{aligned} &[LU].[Ji] = [Ez] \\ \Rightarrow\ &[Ji] = [invLU].[Ez] \end{aligned} \tag{42}$$

In this simple case, every row of the first matrix is multiplied by the single column of the second vector. For every combination, products of the corresponding elements can be done in parallel. Following this, the sum of products must be calculated.

3.2 GPU implementation

In this section, we discuss the implementation of the algorithm's steps on the GPU. We describe how the matrix memory is managed and how the different iterations are distributed over parallel processing elements of a GPU to get an efficient and faster compute time. To review, the overall algorithm is divided into two main tasks. The first task is to populate the impedance matrix *Zmn* for a *p*-point quadrature rule. The values for the incident electric field vector *Ez* on the right hand side of the equation are also computed. The second task is to solve this linear system of equations; this task is divided into three sub-sections: 1) use Crout's algorithm to decompose *Zmn* into lower and upper

triangular matrices stored in *LU*, 2) find the inverse of the triangular matrices in *LU* to give *invLU*, 3) multiply the inverse triangular matrices with the *Ez* vector to obtain the solution.

3.2.1 Task 1

This task consists of generating the *Zmn* matrix and the *Ez* vector using the geometry of the scatterer surface. As mentioned earlier, we read in the (*x*,*y*) coordinates for each node, where two nodes define a cell on the surface of the matrix. Suppose we consider an LCN implementation using a *p*-point quadrature rule. Then, for each observer-source combination, the resulting submatrix is of size p^2. In our implementation, we expand the *X, Y* vectors of size *N* that hold the coordinates of the surface nodes into vectors *Xp* and *Yp* of size *Np* each, that have *p* coordinates for each of the *N* cells. On the GPU, this step is performed by first distributing the input geometry data over *b* blocks, with *t* threads per block. Each thread calculates the *p* coordinates for a specific set of cells. For example, if there are 10 cells in the system, they can be distributed over 2 blocks with 5 threads per block, or, alternatively, they can be distributed over 5 blocks with 2 threads per block. Each thread can then generate its part of the coordinates. In CUDA the grid has a limit on the number of blocks and the number of threads that can be configured. In cases where the problem size is big enough or the number of nodes is large, so that there cannot be a one-to-one correspondence between a thread and a cell, we assign multiple cells to a single thread. We present data on this later to discuss the optimum combination of the number of blocks and number of threads per block. The pseudo-code below describes this process:

Pseudo-code:

---

1. Read in (x,y) coordinates for each node from input file into **X**(N), **Y**(N) and with incident angle ϕ.

2. Allocate memory on the GPU for **Xp_d**(Np) and **Yp_d**(Np).

3. Copy the vectors **X** and **Y** to the device/GPU vectors **X_d, Y_d**

```
4.    Launch device kernel with b blocks and t threads per block
      Expand_XY <<< b,t >>> (*X_d, *Y_d, *Xp_d, *Yp_d)

# in the device KERNEL

5.    Find the unique global id of a thread in block using grid
      dimensions (number of blocks), blockId, block dimenions(number of
      threads in a block), threadId(thread Id within each block)

6.    For (i = cells corresponding to the global ThreadID)

            For (p = number of points in the quadrature)

                  Calculate Xp_d[i + p], and Yp_d[i + p] using values
                  from the p-point quadrature rule.

            End For

      End For

# KERNEL returns to host
```

It should be noted that so far *Xp_d*, *Yp_d* vectors are always stored on the global GPU or device memory and are never copied back to the CPU or host memory as the results stored in them are used in the next step after this kernel returns. Looking at the pseudo-code above, this section is relatively simple to implement on the GPU as there are no dependencies and the amount of memory transfer is minimum.

Now, we are in a position to calculate the *Ez* and *Zmn* matrices on the device using the *Xp_d, Yp_d* vectors. This is done in two separate kernel calls. The first kernel calculates the *Ez* vector. Before launching this kernel from the host, the host code allocates a *Ez_d* vector of size *Np* on the device. The host code then calls the kernel. All elements of the *Ez_d* can be calculated in parallel as there is no dependency between them. In the pseudo-code below, we omit the details for this kernel because of its simplicity.

The next kernel calculates the *Zmn* matrix of the order *Nq*. From the host code, we first allocate the memory for *Zmn_d* matrix. To allow for the complex-valued nature of each element in the matrix, CUDA provides a data type called *float2* which assigns

every element *Zmn_d[i][j]* two components *Zmn_d[i][j].x* and *Zmn_d[i][j].y*, to hold the real and imaginary values, respectively. On the device, the matrix is distributed into smaller chunks that are assigned to each block. As seen in the pseudo-code below, the kernel starts execution by first computing the distance between each observer and source cell in order to determine whether to use the near-field or far-field expression for the matrix entry. Once determined, the thread(s) loops over combinations of *p* points on each of the observer and source cell to calculate the submatrix for this observer – source combination. The *Zmn_d* values have no dependency between them and can be calculated in parallel. Both *Zmn_d* and *Ez_d* matrices are stored on the device and never copied to the host as they are needed for next step in the algorithm. Below, we present pseudo-code for these two kernels.

To illustrate how the problem data is distributed over a grid of processors, we consider a case with 10 cells to describe the scatterer surface and a 3-point quadrature rule used within each cell. When the first kernel is called to compute *Xp_d* and *Yp_d*, we can distribute the problem over 10 blocks with 3 threads per block, since more threads would be redundant. The next kernel that computes the *Ez_d* vector can be launched with 30 blocks and 1 thread per block, 5 blocks and 6 threads per block, etc., as there are 30 entries of *Ez_d* to be computed in parallel. To compute *Zmn_d* we divide the 2D problem into a 5x5 grid of blocks. As each block is required to compute a 6x6 submatrix, we can use 36 threads per block. As an alternative, this could also have been done with a 10x10 grid of blocks each with 9 (3x3) threads per block.

Pseudo-code:

```
1.    Allocate memory on device (N x p x sizeof(float2)) for Ez_d(Np).

2.    Compute_Ez <<< b,t >>> (*Ez_d, *Xp_d, *Yp_d)

# KERNEL details omitted because of simplicity.

3.    Allocate memory on device (Np x Np x sizeof(float2)) for
      Zmn_d((Np)^2).
```

```
4.    Compute_Zmn <<< b,t >>> (**Zmn_d, *Xp_d, *Yp_d)
# in the KERNEL
5.    Compute global threadID using grid values.
6.    For (each observer-cell combination mapped to this thread)
            If (Distance between the observer and source) < LCN_max
                  For (each p points in the observer-source cells)
                        Compute the submatrix Zmn element with LCN
                  End For
            Else
                  For (each p points in the observer-source cells)
                        Compute the submatrix Zmn element with MoM.
                  End For
            End If-Else
      End For
```

3.2.2 Problem Solution: 1. Zmn decomposition into LU

At this point in the algorithm, the *Zmn* and *Ez* matrices have been computed. Now, Crout's method is used to decompose the matrix into an upper and lower triangular matrix. For GPU implemention, we need to consider the dependency between the elements of *LU*. As Figure 7 demonstrates, every element of *LU* is dependent on the values of elements to its left and above it. In the serial algorithm, in the first iteration, we calculate the first row and the first column and in the next iteration we calculate the second row and second column excluding the elements from the first iteration. All together there are *N* iterations in this step for decomposing a matrix of the order *N*.

Consider the computation of element LU(i,j) for the case i=10, j=5. The various factors required for that element are available at different times, depending on the preceding computations. These are illustrated as follows:

LU(*10,5*) =	LU(*10,0*)*LU(*0,5*)	……	Available at 1st iteration
	+ LU(*10,1*)*LU(*1,5*)	……	Available at 2nd iteration
	+ LU(*10,2*)*LU(*2,5*)	……	Available at 3rd iteration
	+ LU(*10,3*)*LU(*3,5*)	……	Available at 4th iteration
	+ LU(*10,4*)*LU(*4,5*)	……	Available at 5th iteration

The above lines suggest that instead of waiting for the sixth iteration to calculate all of the five products and then taking there sum, we could use the results from the first iteration and apply them immediately towards the final value of LU(10,5). In the same way, after every iteration we partially update all the remaining elements of the matrix. This saves considerable amount of time and efficiently uses all the processing power available in a GPU. A version of this has been implemented by Kola et al. on a CPU cluster [12]. Figure 8 shows a logical view of these iterations going over a matrix. Let us also consider the total number of compute cycles it would take to calculate the bottom-right element in *Zmn* since it is calculated at the end and has the longest strand of computations before it gets finalized. When using partial updates, if we assume that there is no cost for inter-block communication or synchronization and there is always one thread per *Zmn* element, then the complete computation can be completed in *N* cycles or *N* iterations. This time is just the cycles it takes to calculate the *N-1* elements above in the same column and the *N-1* elements in the same row to the left of the bottom right element. The result would be available in *N* cycles, but for bigger problems there cannot be a 1-1 correspondence between block threads and elements. Hence the total time is at least $N * (N^2 / (\text{Number of total threads in the CUDA grid}))$. Theoretically partial updates provide massive speedup, but what really costs the most is the inter-block synchronization and data transfer. Together, these three behaviors can be tuned to provide speedup.

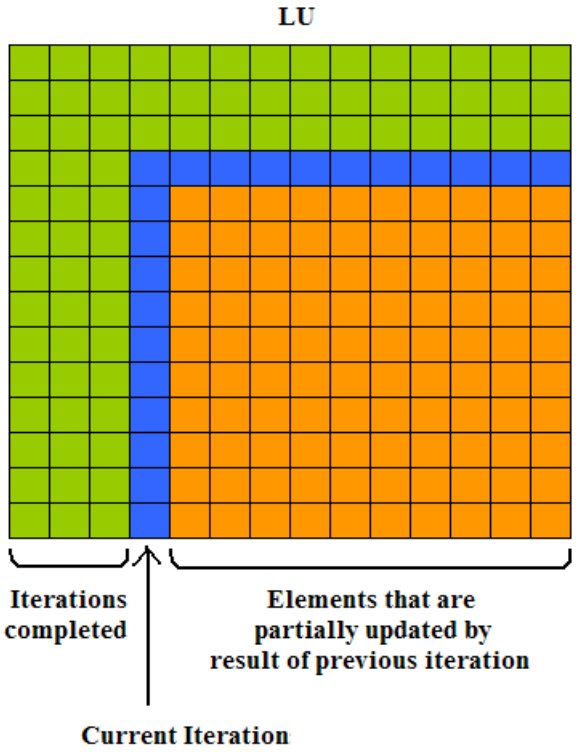

Figure 8 Partial parallel updates in Crout's LU decomposition.

To implement the preceding idea, blocks have to share data and run in a synchronized fashion. CUDA provides intra-block synchronization using a *syncthreads()* directive. This can be used to synchronize all the threads inside a block. For inter-block synchronization, CUDA suggests that one way to synchronize is to use atomic operations on variables located in the global memory as all blocks have access to that location. Atomic operations are implemented such that at any given time, only one thread in the entire grid can modify the value. This is used to implement locks, mutex and other software structures that enable synchronization. However, atomic operations are costly as they take longer time to execute and constrain block and thread schedulers as they serialize their computing while waiting for an atomic operation to complete [10].

Another means of synchronization is to separately call the kernel from the host for each iteration. Unfortunately, the cost of transferring control to and from the kernel is even more expensive. In addition, device hardware would have to load the kernel

instructions into every SM once the kernel call is made — an expensive operation to perform for every iteration.

As a second alternative, Xiao et al. [13] have implemented another version of inter-block communication by using barrier synchronization. As the name suggests, a barrier is a point in a process that all the blocks or threads must reach before they can proceed to the next step. Reference [13] implements this barrier using two arrays labeled *IN* and *OUT*, which are maintained in the global memory so that all the blocks have visibility to their values. The length of these arrays is the same as the number of blocks, as each block has its own synchronization variable or location in the array. Whenever a block completes its computations or iteration, it modifies its corresponding location in *IN*. An arbitrarily chosen lead block is assigned the task of monitoring *IN*. As soon as the lead block sees that all blocks have reached this stage, it modifies appropriate values in the other array *OUT* to indicate that the process may proceed to the next iteration. After the other blocks write to the first array *IN*, they wait for the correct value (next iteration) to appear in the second array *OUT*. Once a block sees the updated value in *OUT*, it knows that all other blocks have reached the same stage and that it can proceed to the next iteration. The maximum number of blocks that can be run in this scheme is limited to the number of SMs that the specific GPU hardware offers. This limit is necessary since an SM can only execute one block at a time, and if a block executing on a SM is waiting for the update from another block which hasn't been scheduled yet as the number of blocks in the grid is large, then the program is trapped in a holding pattern: a situation known as *deadlock*.

We use barrier synchronization in our algorithm by dividing the computation of the *LU* matrices into chunks that are assigned to the blocks, which in turn are each assigned to an SM. The NVIDIA GTX 280 GPU used in this study has 10 SMs which permits 10 blocks in our grid. All thread blocks update their chunk of the *LU* elements as soon as the result from the previous iteration becomes available in the global memory after being computed on a specific block or SM.

There is another optimization that can be made to speed up the overall time. Global memory accesses generally take 200-300 clock cycles to return with the data. Calls to global memory could be made by every block as the results from the previous iteration are needed to modify the values mapped to the current block. For the case of 10 cells in the geometry and a 3-point quadrature rule, with a grid made up of 5x5 blocks, the mapping is as shown in Figure 9. Here, block (4,2) accesses the highlighted elements that were finalized in the previous iteration to partially update its sub-matrix. These values are used multiple times to update all the elements in the sub-matrix, and if they were accessed using a global read instruction the overall speed would suffer. As mentioned earlier, every SM has a local cache known as Shared Memory in the CUDA programming model to store data to be used within a block's thread. Read accesses to this shared memory are approximately 150 times faster than to global memory. Therefore, in our implementation, every block copies the finalized values from the previous iteration only once into the shared memory. All threads read from the shared memory to update their set of elements of the block's shared memory. This saves considerable compute time by avoiding multiple reads of the same value from global memory.

Figure 9 Shared Memory optimization.

Below, we show the logic flow for these steps in pseudo-code. Another aspect of our implementation is the use of the *__syncthreads()* instruction to synchronize within each block. For example, at line 5 the block threads copy data from the global memory into the shared memory. Now the total number of elements in the sub-matrix are 36 (6 by 6), the same as the number of threads in the block. The total number of finalized elements from the previous iteration are 12 (6+6). This means that only 12 out of 36 threads can be used to copy the global data into shared memory. To determine which threads to use we check the local thread identification (*threadid*) and only these threads make the global memory access. If *threadid* is more than 12, that thread will skip the copying code and start updating its assigned elements in the sub-matrix using invalid values in the shared memory. To avoid this, the *__syncthreads()* instruction, which acts as a local barrier inside a block, is used to make the non-copying threads wait for the data to be copied. We now present the pseudo-code for these steps (part of the intra-block synchronization is taken from [13]):

Pseudo-code:

```
1.    Allocate memory on device (Np x Np x sizeof(float2)) for
      LU_d(Np^2). Also allocate global arrays IN_d(number of blocks),
      OUT_d(number of blocks)

2.    Compute_LU <<< b,t >>> (**LU_d, **Zmn_d)

#in the kernel

3.    For (n = number of iterations)

4.          #### Update values from the previous iteration within a
            block

5.          if(localThreadId <= number of global elements to copy)
                  Copy finalized values from the previous iteration
                  into shared memory of the SM.

6.          __syncthreads();  # to avoid other threads from trying to
            access the shared memory before they are populated with the
            new values

7.          Calculate global threadID to map matrix elements to a
            thread
```

```
8.          Partially update the corresponding values.

9.          #### Inter-Block synchronization starts here

10.         If(localThreadID == 0) # Only thread 0 used for
            synchronization

11.               IN_d[blockId] = iteration number;

12.         End IF

13.         #### Using the lead block to check the status of all blocks

14.         if(blockId == 0)

15.               If(localThreadID < Number of Blocks)

16.                     while (IN[localThreadId] != iteration);   #
                        Busy-wait on the blocks to reach barrier

17.               End If

18.               __syncthreads(); # To prevent threads of the lead
                  block from modifying the OUT values.

19.               If(localThreadId < Number of Blocks) # Signal all the
                  other blocks to continue computing for next iteration

10.                     OUT[localThreadId] = next iteration;

11.               End If

12.         End If

13.         #### Every block busy-waits on the corresponding OUT value
            to be equal to the next iteration to continue computing

14.         If(localThreadId == 0)

15.               while(OUT[blockId] != next iteration);

16.         End If

17.         __syncthreads();  # Make all threads in a block wait for
            OUT value to be reached

18.   End For
```

3.2.3 Problem Solution: 2. Inverse of LU matrices

Following the computation of *LU*, we can determine the inverse of the lower and upper triangle simultaneously, as they are completely independent of each other. Consider the dependency in the first few iterations in finding the inverse of the lower triangular matrix:

@ j = 1 (43)

```
invLU(2,1) ← invLU(1,1)
invLU(3,1) ← invLU(1,1), invLU(2,1)
invLU(4,1) ← invLU(1,1), invLU(2,1), invLU(3,1)
     :
     :
     :
```

@ j = 2

```
invLU(3,2) ← invLU(2,2)
invLU(4,2) ← invLU(2,2), invLU(3,2)
     :
     :
     :
```

As seen in the rolled out iterations above, there is no dependency among columns (*j*) or the outermost for loop in the pseudo-code. But, within each column iteration there is a dependency on the previous iterations or the elements above. Therefore, each section of the inverse calculation can have parallelism to the N^{th} order (number of columns). Figure 10 illustrates that approach.

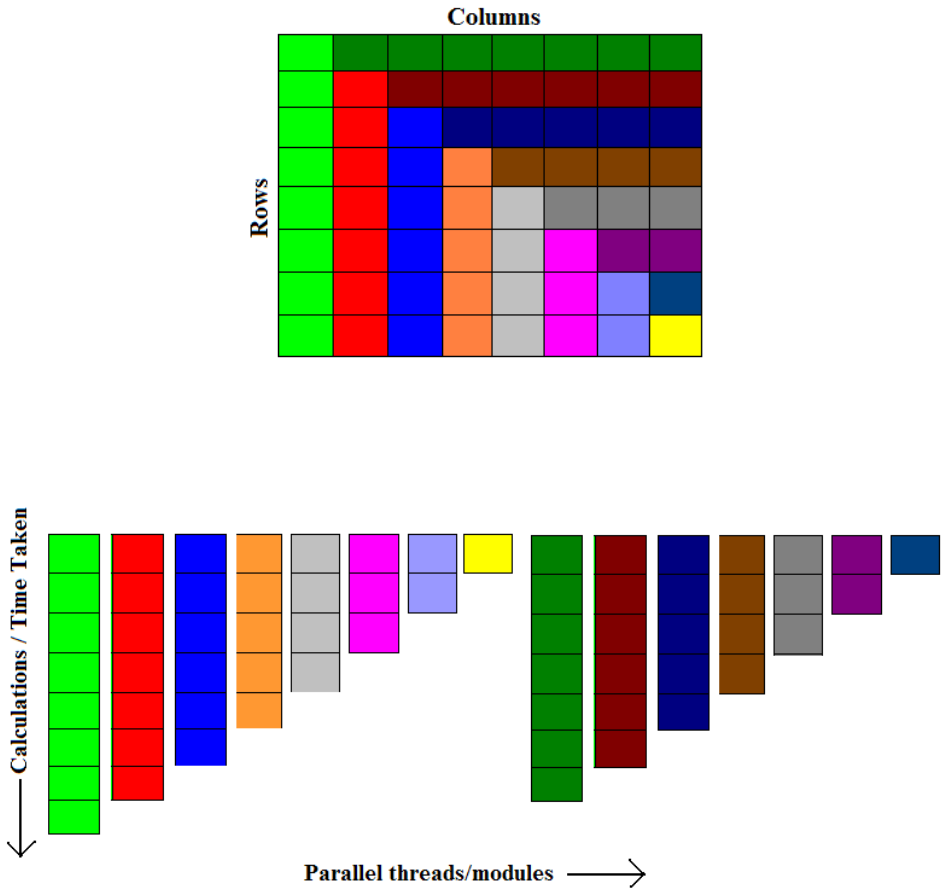

Figure 10 Parallelism in computing the inverse of the lower and upper triangle.

In Figure 10, the rows and columns of the LU matrix are spread out over different parallel execution cores or modules. All of these threads can execute in parallel with no dependence on each other. It should be noted that the threads do not have same execution times. For example, the longest thread here in light green has an execution time eight times that of the smallest unit, which is in yellow. This means that the smallest thread will finish its calculations and then wait or idle its processor core waiting for the longest thread to finish. A more efficient use of the computing modules would be to combine these threads in such a way that they have the same execution time. This way the time spent by one core waiting for others to finish is minimal. This is even more advantageous when the number of parallel threads is more than the number of available processor cores. Figure 11 illustrates that approach.

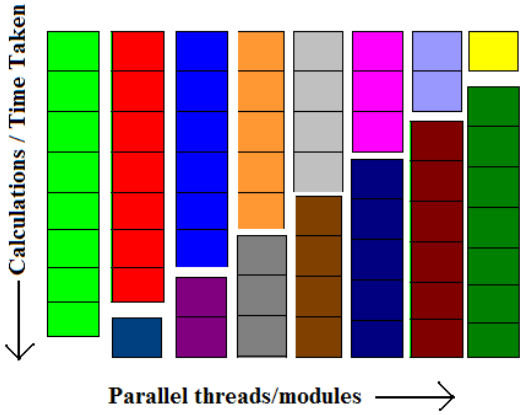

Figure 11 Efficient use of parallel threads.

We also apply the concept of partial updates to this step of the algorithm. As seen in Eq. 43, the data finalized by the first iteration can be used to update the values of the other elements in the same column for the lower triangular matrix and for the same row in the upper triangular matrix. But a big difference in this implementation compared to the one in the last section for computing *invLU* from *LU* is that the data is not shared between blocks. As no column in lower triangular matrix or a row in upper triangular matrix is distributed across more than one block, there is no dependency between blocks. So, for any specific block, which may have multiple columns or rows assigned to it, with dependencies existing within columns, we spread the calculations over multiple threads. For example, if a block is responsible for calculating the values for the 10 elements in a column, then we could distribute the calculations over 5 threads, where each thread is responsible for a chunk of two elements. For partial updates, during the first iteration, only thread 0 finalizes the value of first element in the column. In the next iteration, all threads can use that first element's value to modify the values of elements they are responsible for. In the serial approach, the computation time is proportional to $O(N^3)$, while the parallel GPU version executes in a time proportional to $O(N^3/p)$, where p is the number of parallel computations possible. The following operations explain this process in more detail:

```
Iteration 1:
      Thread 0:   Computes Column[0]
      Threads 1-4: No computation
```

```
Iteration 2:
      Thread 0:   Computes Column[1] using Column[0]
      Thread 1:   Updates Column[2] using Column[0]
                  Updates Column[3] using Column[0]
      Thread 2:   Updates Column[4] using Column[0]
                  Updates Column[5] using Column[0]
      Thread 3:   Updates Column[6] using Column[0]
                  Updates Column[7] using Column[0]
      Thread 4:   Updates Column[8] using Column[0]
                  Updates Column[9] using Column[0]
Iteration 3:
      Thread 0:   Nothing else to compute.
      Thread 1:   Computes Column[2] using Column[1]
                  Updates Column[3] using Column[1]
      Thread 2:   Updates Column[4] using Column[1]
                  Updates Column[5] using Column[1]
      Thread 3:   Updates Column[6] using Column[1]
                  Updates Column[7] using Column[1]
      Thread 4:   Updates Column[8] using Column[1]
                  Updates Column[9] using Column[1]
Iteration 4:
      Thread 1:   Computes Column[3] using Column[2]
      Thread 2-4: Update there elements using Column[2]
      . . .
```

We now present the pseudo-code for this step:

```
1.    For i 0→N-1                          # INITIALIZATION
2.          For j 0→N-1
3.                invLU(i,j) = 0
4.          end for
5.    end for
# Kernel call
6.    For iter 0→NumberOfIterations        # LOWER TRIANGLE
```

```
7.          For i startThreadRow→endThreadRow
8.                invLU(i,j) = invLU(i,j) – LU(I,iter)
9.          End For
10.         __syncthreads();
11.   End For
12.   Same steps for Upper Triangle
```

3.2.4 Problem Solution: 3. Multiplication of invLU with Ez

This step consists of multiplying the inverse of the lower and upper triangular matrices with the *Ez_d* column vector that represents the right hand side of the equation. This implementation requires the multiplication of every row of the inverse triangular matrices (first the lower and then the upper) with the corresponding element in the column vector. Since this step is heavily based on sum of products we implement a tree flow on the GPU. First, for a specific combination of a row and a column (*Ez_d* vector) within a block, the products are distributed among all the threads, the results from which are stored in the shared memory. Once products are taken, the results are distributed among half the number of threads to be added. Once this step of addition is done the results are added again using half of those threads, and this continues until all the products are summed into one value. Figure 12 shows this process for a case of 10 elements in row. Results presented in the following chapter show that the execution time of this task is less than 1% of the complete algorithm, even for a problem as large as 5000 nodes. For this reason, we omit a discussion of the GPU implementation of this step.

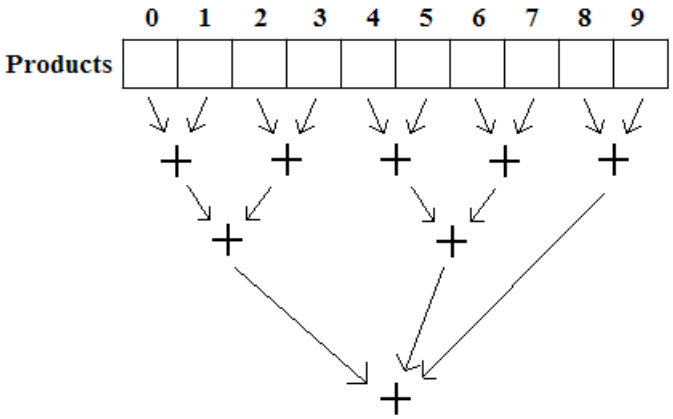

Figure 12 Parallel Sum-of-Products on a GPU.

4. RESULTS

In this section, we present data collected by running simulations in several different configurations. As a baseline for comparison, we consider two single-threaded versions. First, the entire problem is executed in MATLAB on a Windows laptop containing a single core Intel Pentium M processor running at 1.6 GHz (2004) with 1GB of physical memory (RAM). Second, the exercise is repeated using a Linux system containing an Intel Core 2 Quad CPU Q9550 running at 2.83GHz with 2GB of physical memory (RAM). Finally, we repeat the test using the second (Linux) system but augment the processor with an NVIDIA GTX280 graphics card to create a multi-threaded environment. The graphics card has the following system parameters:

Number of Streaming Multiprocessors (SM):	30
Number of Cores (8 per SM):	240
Total Global Memory:	1GB
Total Shared Memory Per Block:	16KB
Warp Size:	32
Max Threads per Block:	512
Maximum Dimension of a Block:	512 x 512 x 64
Maximum Dimension of a Grid:	65535 x 65535 x 1
Clock Rate:	1.3 GHz

4.1 MATLAB implementation

In this test, we use the MATLAB language to benchmark the performance of the overall solution algorithm. The steps taken in MATLAB are as follows: 1) Read in the *X* and *Y* coordinates for the geometry. 2) Calculate the impedance matrix *Zmn* and the excitation vector *Ez* using the same algorithm that is implemented in C elsewhere and described above in pseudo-code. 3) Solve the system using internal MATLAB functions to compute the result. This allows MATLAB to choose the optimum method implemented within its software package. Figure 13 shows the execution time as a function of the order of the system of equations. The horizontal axis in the execution time graph is the number of nodes (unknowns) and the vertical axis is the total execution time. In the right hand plot of Figure 13, the magnitude of *Ji* is plotted around the scatterer

surface for the case where the scatterer has a circular contour of radius one wavelength and the incident electric field has unity magnitude (1 V/wavelength).

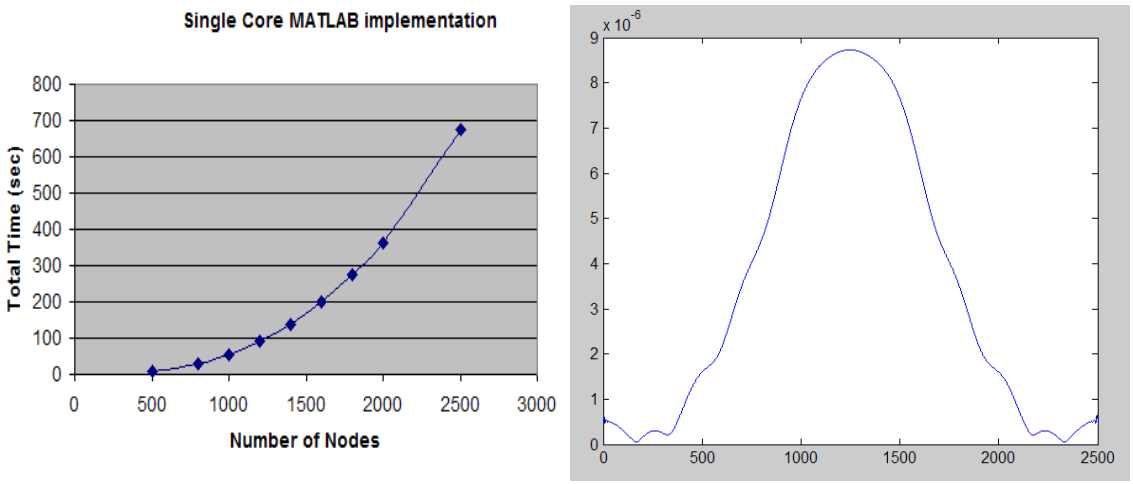

Figure 13 a. Execution time for the MATLAB implementation using a single core processor. **b.** Magnitude of the current density obtained as the solution for the 2500 unknown problem, plotted as a function of node number.

As expected, the overall time taken by this algorithm is approximately $O(N^3)$ where N is the number of nodes in the problem. The factor of N^3 is characteristic of matrix solution algorithms, and suggests that the overall time will increase by a factor of 8 when the number of unknowns doubles. Since a 2500 node geometry requires 700 seconds (~12 minutes), a problem with 5000 nodes would take approximately 5600 seconds (~93 minutes) and a problem with 10,000 nodes would require more than 12 hours.

4.2 C Language Implementation

In this section, we implement the MoM/LCN algorithm in the C language and study its execution time using a quad core processor. Although the processor contains four cores, this single threaded implementation executes in a serial fashion using only one of the cores of the machine. In contrast to the MATLAB implementation, every step of the C language algorithm was programmed by the author. This provides a baseline to compare the serial CPU and parallel GPU implementations for each part of the algorithm. The resulting execution times are shown in Figure 14.

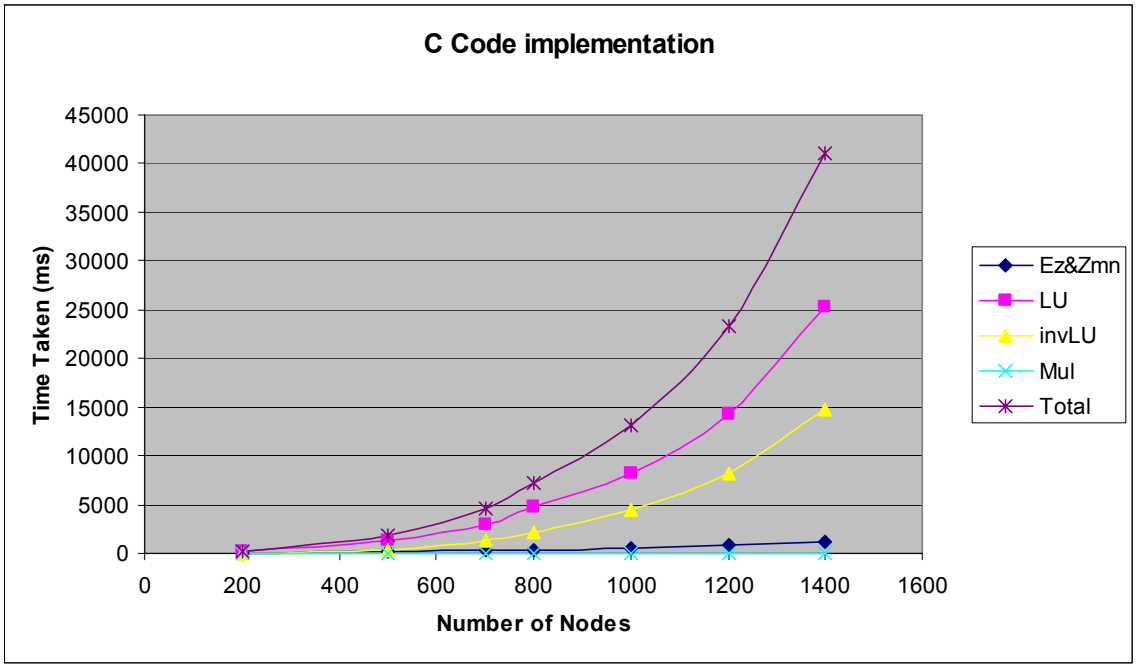

Figure 14 Single threaded C code implementation of the algorithm.

Figure 14 shows the time taken for the completion of each part of the algorithm. Decomposing the matrix into lower and upper triangles using the Crout LU algorithm is an $O(N^3)$ process, in common with the internal MATLAB routines. Inverting each triangular matrix is also an $O(N^3)$ process. This can be observed from the pseudo-code in the previous chapter.

4.3 GPU Device

In this section, we examine the performance improvement achieved by the parallel implementation of each part of the solution algorithm using the GPU. As the previous sections described, each step of the algorithm can be optimized. For instance, the combinations of block and thread counts can be varied, the amount of shared memory in use can be adjusted, and partial updates can be obtained during each iteration. We analyze data for each of these optimizations and attempt to determine optimum configurations. We also discuss the weaknesses in the implementation and places where more improvements can be extracted.

A baseline performance can be obtained by executing the algorithm as a single thread on the GPU. Figure 15 shows the execution times of each part of the algorithm versus the number of nodes in the model. These data give an upper bound on the capability of a GPU for computations where thousands of these threads can be running simultaneously, each delivering performance shown below.

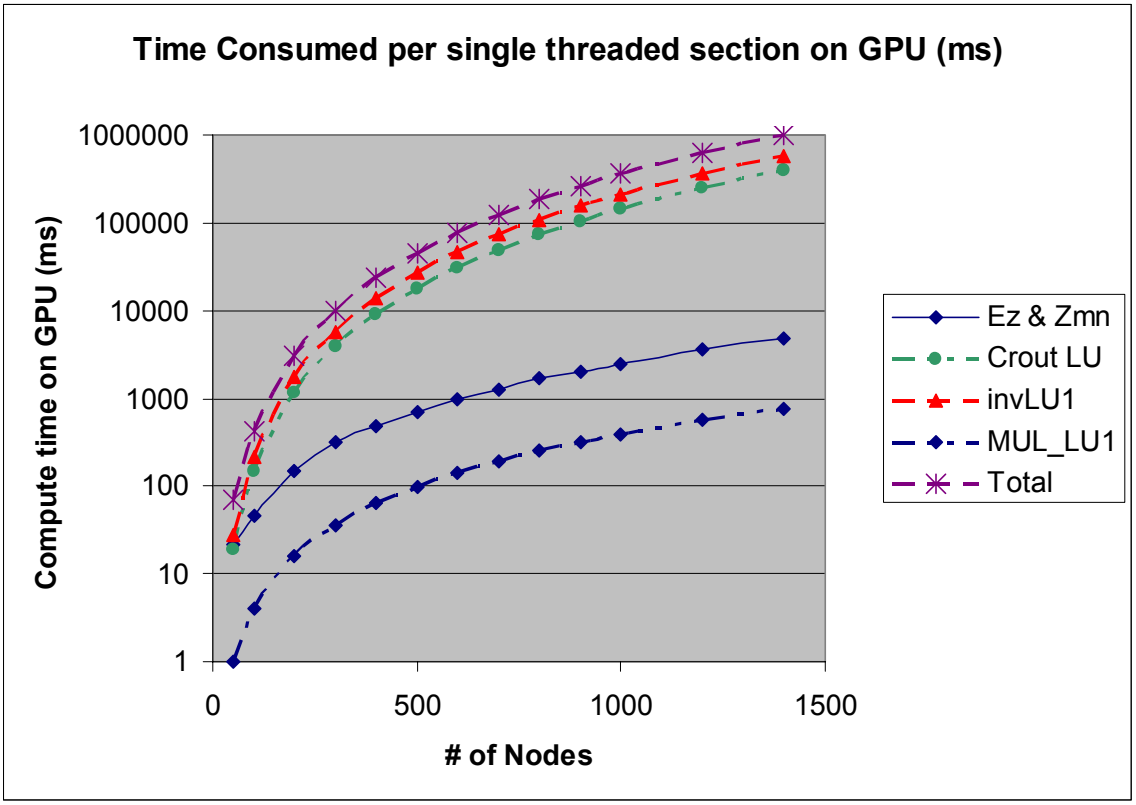

Figure 15 Time consumed in a single threaded implementation on GPU.

4.3.1 Generating Ez and Zmn matrices

In this section, we examine the performance improvement achieved by the parallel implementation of the first step on the GPU, the computation of *Ez* and *Zmn* matrices for a given model of a scatterer. Since this step involves no dependency between different elements of the matrices, it harnesses the parallel compute power of the GPU very well. This part is divided into three kernel calls. The first kernel computes the *p*-points of each cell's *X* and *Y* coordinates at the quadrature nodes to be used in computing

Ez and *Zmn*. The next two kernels compute *Ez* and *Zmn*. In a problem with *n* cells on the scatterer and a *p*-point quadrature rule in each cell, the first kernel produces *np* values that are used by the next two kernels. The *Zmn* matrix is of order $N = np$. For the second and the third kernels, the amount of computation depends on the total order *N*. As expected, the majority of the computation time in this step comes from calculating the *Zmn* matrix because the *Ez* vector is only one dimensional.

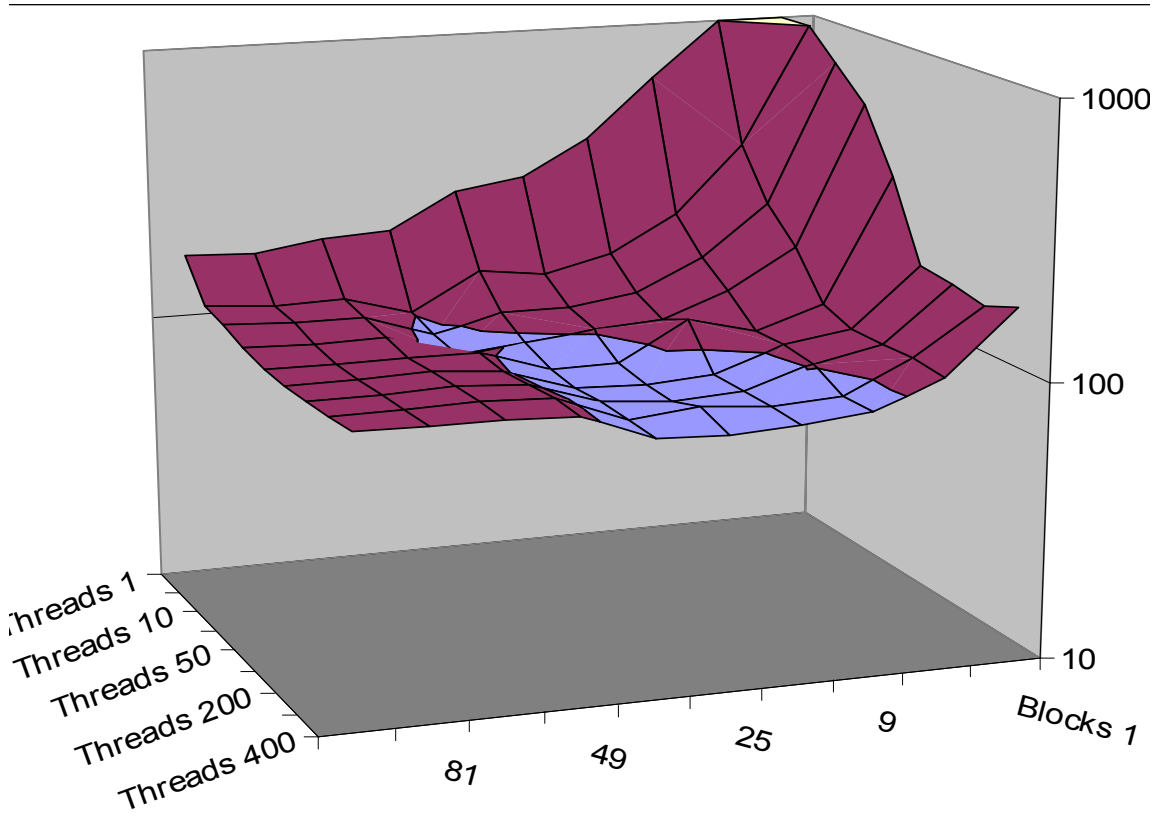

Figure 16 Time taken for different combinations of threads and blocks.

The steps of the algorithm each require some optimization. As an example, Figure 16 illustrates the GPU execution time for various combinations of grid sizes for a problem of 1000 nodes. These results show that even though the CUDA grid allows a maximum block grid of 65535 by 65535 by 1 and a maximum of 512 by 512 by 64 threads per block, the optimum configuration depends on the characteristics of the kernel. In Figure 16, the optimum combination is approximately 9-25 blocks and 150-250 threads per block. It should be noted here that the execution time shown in the plot

includes the time to transfer the instructions from the CPU to the GPU, and the time to schedule the instructions or a "warp" of the kernel on each SM before the actual instructions start computing. Hence, the optimum configuration will be different for each kernel and the amount of data computed.

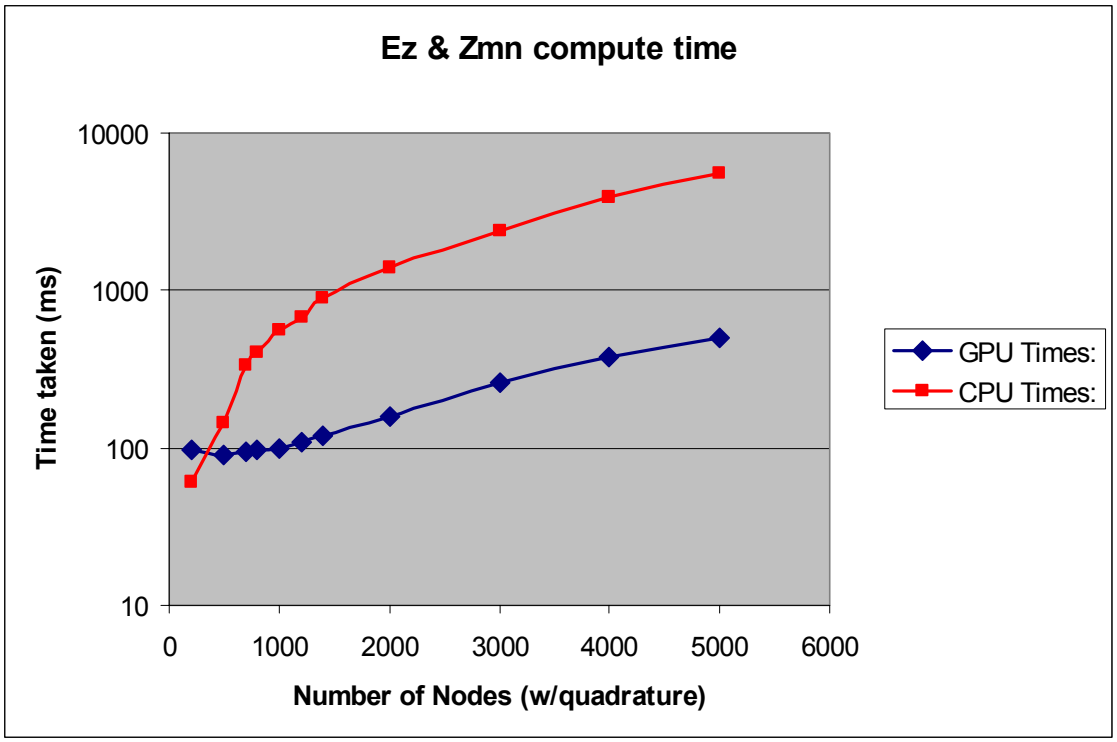

Figure 17 Comparison of CPU and GPU implementation of *Ez* and *Zmn*.

Figure 17 shows the execution time arising from the generation of the *Ez* and *Zmn* matrices versus the number of nodes. The figure shows two curves; the first one in red represents the time taken in the computation of the *Ez* and *Zmn* on the CPU in the serial fashion, while the second curve in blue plots the time taken to compute the same matrices using the GPU. The GPU implementation is faster, with an increase in speedup as the problem size increases. For smaller problems where the GPU's parallel compute power is not completely utilized (a few hundred unknowns) the CPU implementation is able to provide respectable performance. But as the matrix size increases, the GPU's cores are better utilized and the CPU reaches its maximum performance.

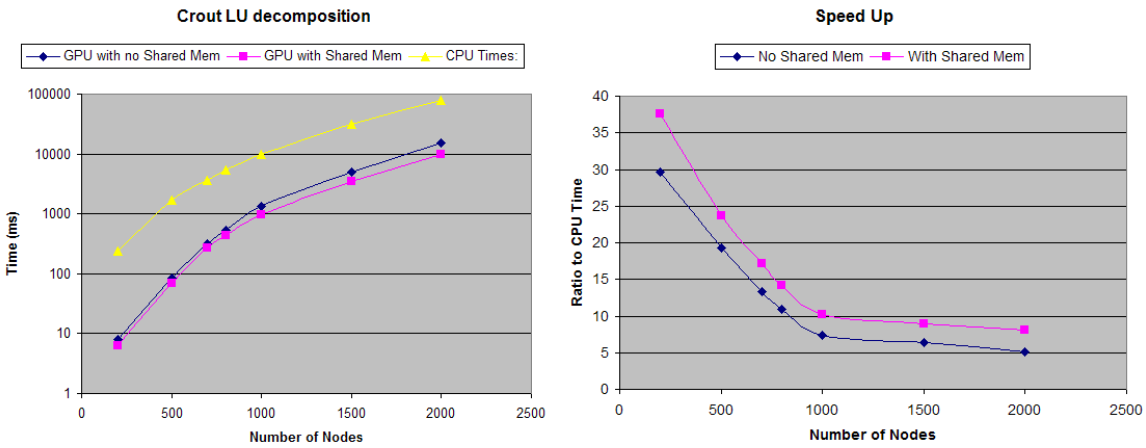

Figure 18 Parallel version of Crout's algorithm without using shared memory.

4.3.2 Computing Lower & Upper Triangular matrices (LU)

In this section, we examine the performance improvement achieved by the GPU implementation of Crout's LU factorization algorithm. Figure 18 shows the data collected to analyze the performance. The GPU implementation is faster, with an overall speedup that saturates at about a factor of 7. One reason for the saturation is that the maximum configuration is limited to 30 blocks with a maximum of 512 threads per block. Beyond a point the only way to improve performance would be to allow more blocks and threads. However, the major limitations this implementation suffers from are the synchronization required among blocks and the amount of data shared between blocks. These two limitations may not improve as the number of processors is increased because the amount of communication needed among the processors becomes a bottleneck. Overall performance improvements come from using partial updates. Consequently, the final computation is delayed, but by much less than in the serial version. As we discuss further in this section, however, the overall efficiency of the LU factorization is primarily memory bandwidth limited.

As noticed in the plots in Figure 18, the use of shared memory provides some improvement. The improvement is obtained when the result of the previous iteration is stored in the shared memory assigned to the block, instead of in the global memory. This allows the individual threads to update their corresponding data from local memory, which saves numerous clock cycles compared to accessing global memory. The amount

of shared memory offered by the current GPU architecture is 16KB per streaming multiprocessor (SM). The results presented above use all this memory to enhance performance by storing a maximum of 2000 elements. Equation 44 illustrates the mapping of 2000 elements in the shared memory, where each complex value is stored as two floating point values.

$$2000 \text{ elements} \times 2 \frac{\text{floating point values}}{\text{Element}} \times 4 \frac{\text{bytes}}{\text{float}} = 16000 \text{ bytes} = 16 \text{ KB} \quad (44)$$

Figure 19 shows the amount of memory transfer per block taking place as a function of the problem size and the number of blocks throughout the complete simulation. This is dependent upon the number of elements that each block is assigned and the number of times these values have to be read from, updated, and stored back to the global memory. The memory transfer grows quadratically with the problem size. Having more streaming multiprocessors provides some relief to the number of reads and writes as each block is assigned to fewer elements. However, as the order of the problem increases, the number of iterations or the number of times these values have to be updated grows. This gives us a measure of the dependence on memory bandwidth. The greater the number of memory accesses, the longer it will take to finish the simulation. The synchronization process is affected, since the latest global state of all blocks is stored in global memory and slower blocks delay this process.

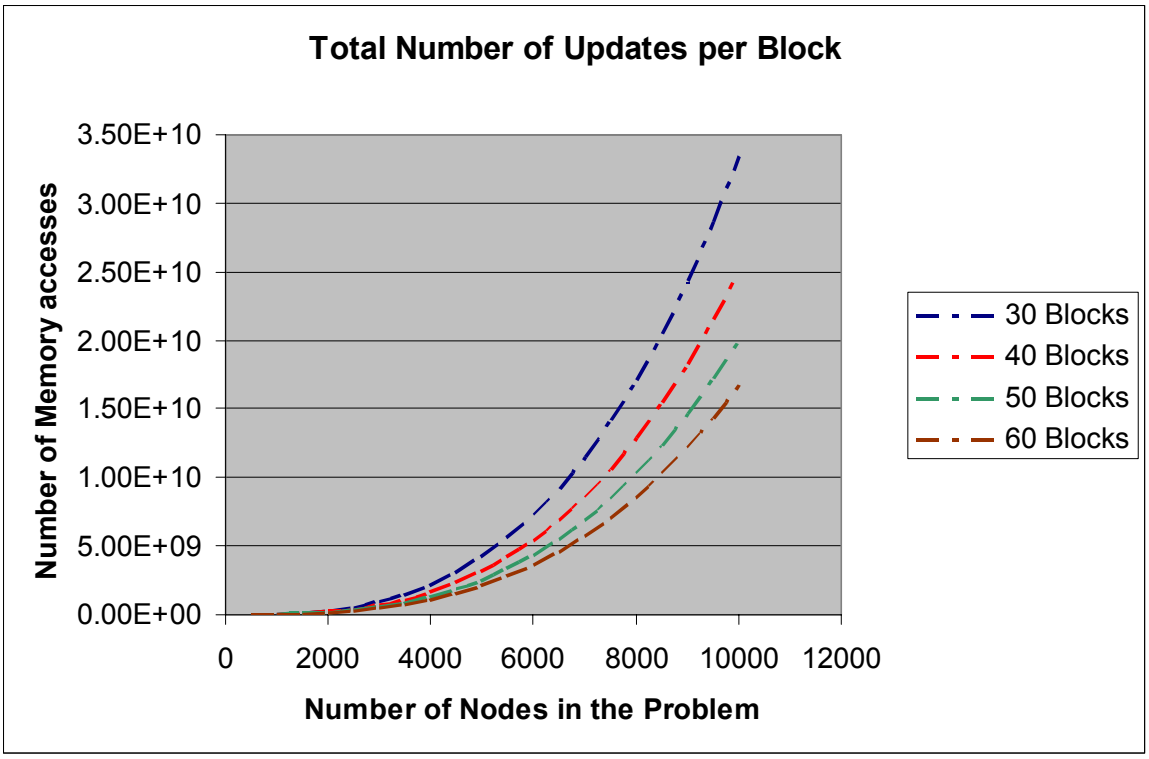

Figure 19 Number of partial updates with different configurations.

Next, we consider the effect of shared memory size per streaming multiprocessor. Figure 20 shows the number of global memory accesses that are made as a function of problem size and shared memory size. The memory accesses made by any block can be of two types. The first consists of reading, updating and writing back data that a block is responsible for, while the second involve updating values that were modified in the previous iteration in a separate block. Shared memory is used to store the second type since these values are used multiple times to update every element in a row or column of a block. Obviously, all these elements cannot be stored in the shared memory as there is only limited amount of it available. We vary the size of shared memory available and then plot the number of elements of the second type that would have to be accessed from the global memory times the number of iterations. An observation that can be made is that if there are 64KB of shared memory available, global accesses only occur for problem sizes of 8000 or larger. The formula used for these updates is given in Equation 45:

$$\begin{array}{lcll} \text{\# of Global} & = & \text{\# of elements in the block} & \text{x} \quad \text{\# of iterations} \\ \text{Mem Accesses} & & \text{that use updates from} & \\ & & \text{Global Memory} & \end{array} \qquad (45)$$

Figure 20 suggests that shared memory lowers the overall simulation time, since accessing results from shared memory is far less time consuming than accessing them from global memory.

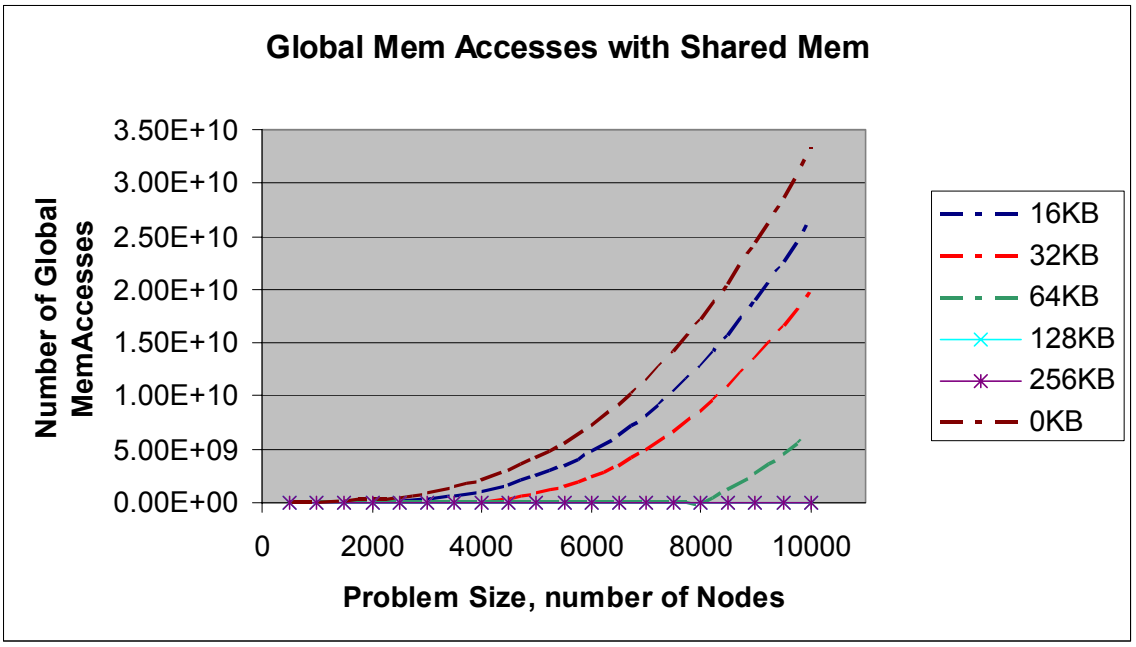

Figure 20 Number of global memory accesses for results of previous iterations.

4.3.3 Computing the Inverse of the Upper & Lower matrices (invLU)

In this section, we examine the performance improvement achieved by the parallel implementation of the step that computes the inverse of the triangular matrices on the GPU. As the previous sections described, in this method we distribute complete columns to blocks because any element in the column is dependent on the values above it. Another optimization implemented in this step is the partial update of the elements in a column as the iterations proceed. Unlike the *LU* factorization where a fixed number of blocks were used because of block synchronization, this step can use any value as the columns are divided among the blocks and there are no results shared between them.

For example, in a configuration involving an *LU* matrix of order 100, when using 20 blocks each block gets 5 columns that are then divided among its threads. ("Columns" here refers to the columns of the lower triangular matrix plus rows of the upper triangular matrix, as any given block is assigned a row and column, as was shown in Figure 11.) We therefore first analyze the nature of the kernel by implementing different combinations of blocks and threads. Figure 21 shows the performance for the case of 1000 nodes. It is observed that there is no obvious optimum combination. This means that there is more than one factor that dictates the performance of this kernel on different configurations. In one corner of the plot, for fewer blocks, poor performance can be attributed to the GPU being underutilized. Thus, execution is the bottleneck instead of memory bandwidth. As the number of blocks and threads are increased, the performance improves as more execution is taking place. But as these numbers continue to increase, beyond a point we observe diminishing performance. This optimum point in the plot below is located on the diagonal, a region where the execution time remains mostly the same (~250 msec). With additional increases in the number of blocks and threads, other factors become a bottleneck. One such bottleneck is the cost of scheduling warps or threads of a block to different cores within the same SM. Another bottleneck could be due to the amount of time a thread has to waits for its read to return from the global memory. This time could be high if there are multiple threads waiting on the same location in the global memory. For this reason, implementing too many threads can create a bottleneck due to memory bandwidth — one of the reasons for spike in the corner of the plot with high number of blocks and threads.

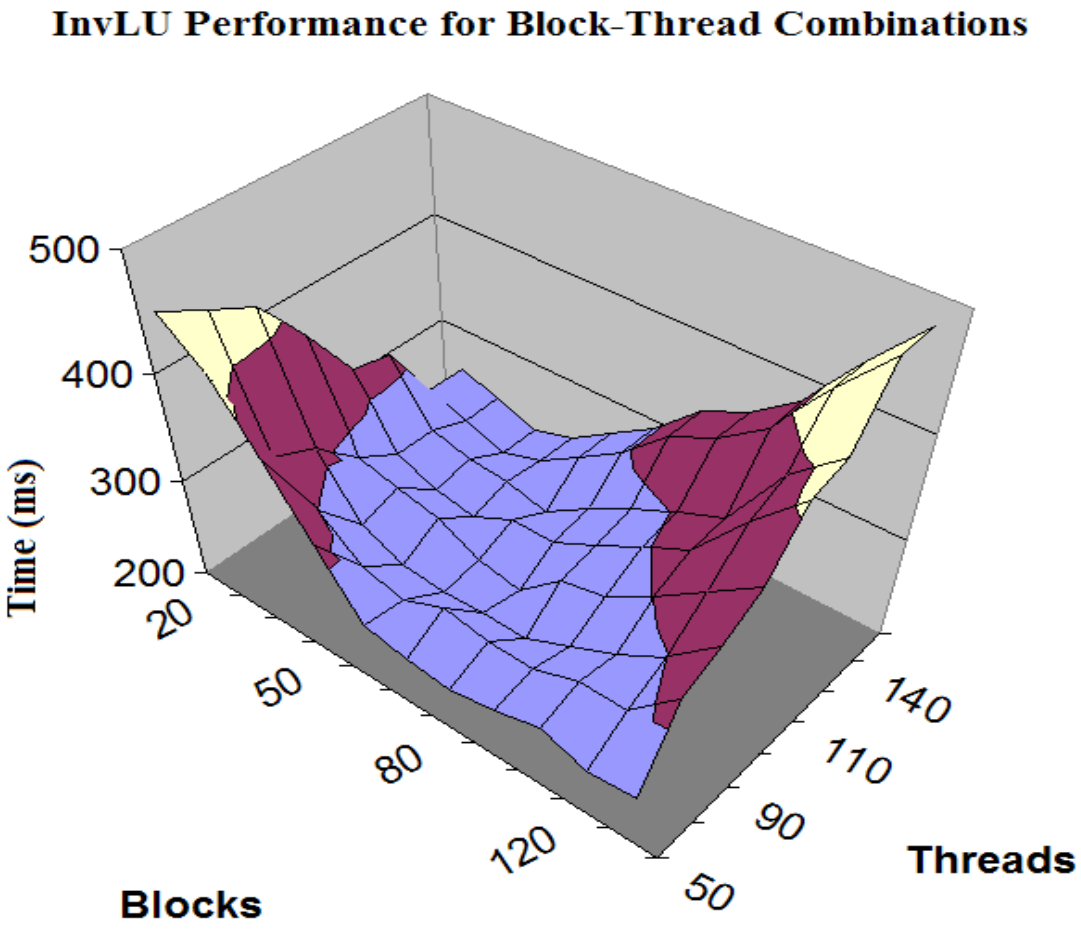

Figure 21 Dependence on number of blocks and threads.

Figure 22 shows the speedup achieved over different problem sizes when computing the inverse of the triangular *L* and *U* matrices. It is observed that the speedup continues to grow until the problem size reaches 4000 or 5000 nodes where it saturates to approximately a factor of 35.

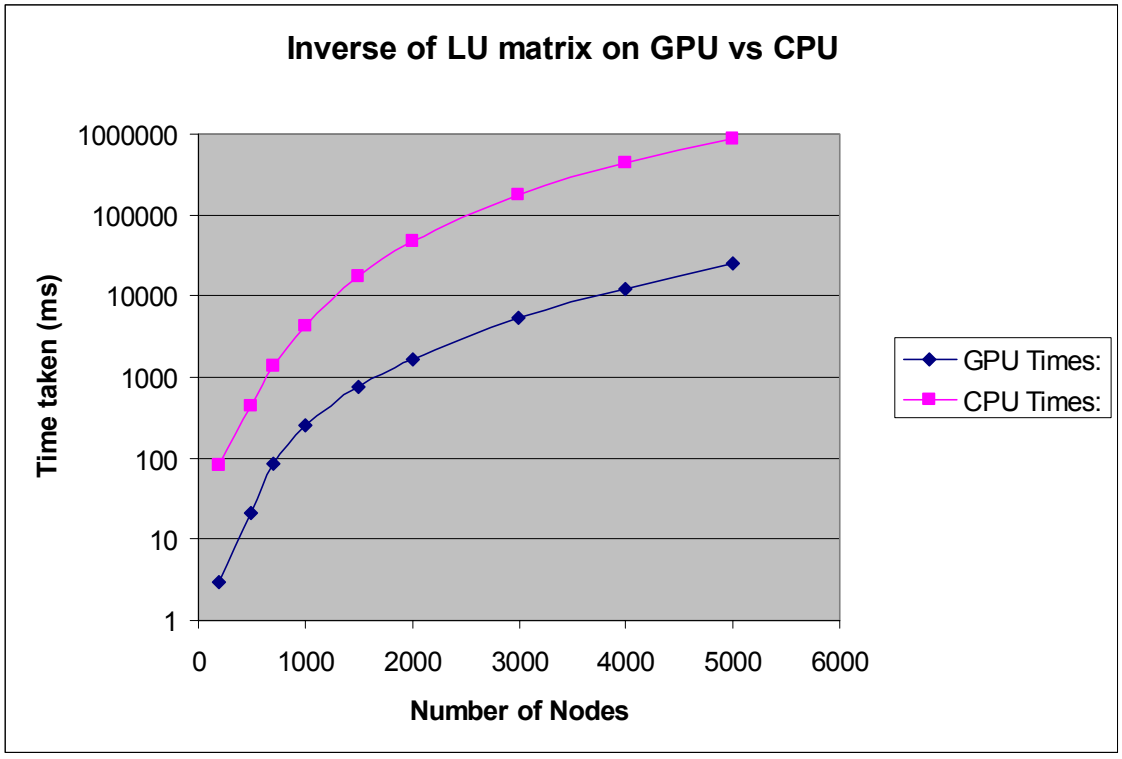

Figure 22 Performance of inverse LU step on GPU Vs. CPU.

4.4 Results Summary

In this section, we examine the overall performance improvement achieved by the GPU implementation. We sum the computation time for each of the above steps and compare it to the total time of the CPU implementation. Figure 23 shows the performance. Out of the three steps of 1) calculating the *Ez* & *Zmn* matrices, 2) Crout's *LU* decomposition, and 3) inversion of matrices *L* and *U*, the second step has the least improvement when implemented on the GPU. This is a consequence of the characteristics of the kernel: first, there can only be as many blocks are there are SMs and, second, the blocks have to synchronized and share data. Nevertheless, the GPU version outperforms the CPU implementation. The performance improvement saturates at an approximate factor of 10 as the problem size grows. Shared memory is used to optimize to another level but it quickly becomes insignificant as the problem size increases and the memory is just not large enough to contribute in a huge way.

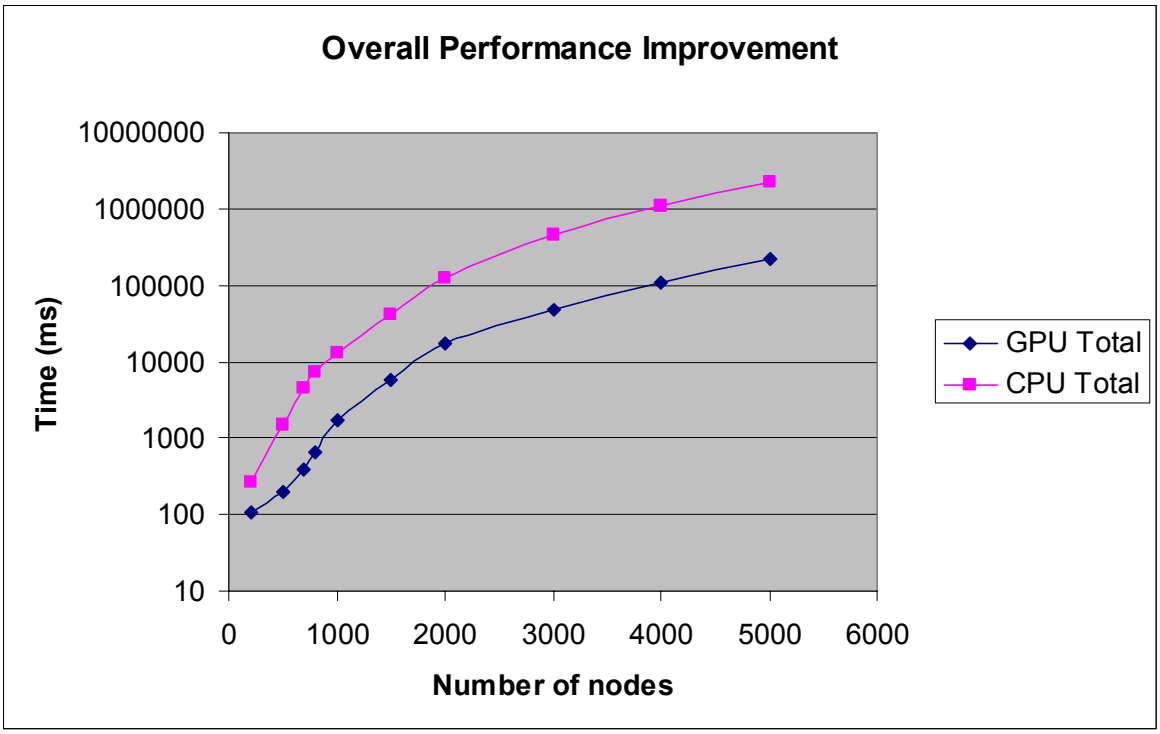

Figure 23 Overall performance of the algorithm GPU vs. CPU.

5. CONCLUSIONS

The aim of this thesis was to describe a parallel-computer implementation of the Method of Moments (MoM) and Locally-corrected Nyström (LCN) algorithms for a simple electromagnetic scattering problem, using a GPGPU and the NVIDIA CUDA interface as the parallel processor. The MoM and LCN procedures were described, as were the GPU architecture and the CUDA programming model. Detailed steps of the MoM/LCN algorithms were explored in order to identify strategies for their parallel implementation. Finally, serial and parallel versions of the algorithm were tested.

We divided the complete algorithm into four steps, where the first one generates the incident electric field *Ez* vector and the impedance matrix *Zmn* using input data that describes the surface of the scatterer. The second step uses the Crout's LU method to decompose the *Zmn* matrix into lower and upper triangular form. The third step finds the inverse of the lower and upper triangular matrices, and those are multiplied with the *Ez* vector in the fourth and the last step. From the point of view of implementing these steps on the GPU, we find the first step is the easiest to implement in parallel as there is no dependency between any two values in either *Zmn* or *Ez*.

The second step, factoring *Zmn* into triangular matrices *L* and *U*, involves data dependencies that make the parallel implementation difficult. When the *Zmn* entries are divided into different blocks that are assigned to separate streaming multiprocessors, the factorization algorithm requires the result of the computations on one block to be shared with many other blocks, meaning that data must be transferred between blocks and the computations must be synchronized. Synchronization is an expensive process that hampers the overall performance. The large amount of data sharing causes the algorithm to be constrained by the available memory bandwidth. Some improvement was observed by the use of shared memory to lower the amount of memory transfer. Despite these limitations, a speedup of about a factor of 7 was obtained for the LU factorization. In this case, the speedup is primarily from the implementation of partial updates.

Note that the CPU core frequency is 2.8-3.5GHz, whereas the GPU core frequency is 1.3-1.5 GHz. Since the CPU is roughly twice as fast in doing the same computation, the actual speedup due to the parallel (GPU) implementation of the LU factorization is about 14. This speedup would be improved if there were more Streaming Multiprocessors in the GPU. NVIDIA just released the new 'Fermi' architecture of GPUs. The new GTX480 card has 480 CUDA cores distributed among SMs, compared to 240 in the GTX 280 card used in the present study. Another feature that would improve the performance is that each SM on the new card will have 48KB of shared memory instead of just 16KB. The new cards have other enhancements such as multiple store/load units per SM. NVIDIA has also introduced the "Gigathread" engine with which multiple blocks can run concurrently on the same SM. Considering these and other features such as better memory bandwidth, a 15x performance is expected with the new hardware as compared to the 7x observed here.

The third step of the MoM/LCN solution algorithm calculates the inverse of the triangular matrices. In this step the data dependency can be kept within each block, eliminating the need for synchronization or data transfer between blocks. As a consequence, this step is much faster than the second step and a speedup of approximately 35 was observed for problems of order 5000.

The final step of multiplying the inverted triangular matrices was not implemented on the GPU, since its contribution to the total run time is less than 1% for the range of problem sizes under consideration.

For the overall MoM/LCN procedure, we observed a total speedup of approximately a factor of 10. The overall result includes the combination of the factor of 7 from the second step with the factor of 35 from the third step, and so on.

Future implementations

This study suggests several ideas to speed up the overall performance of the matrix solution algorithm. The most time consuming step is the Crout LU decomposition.

The data suggest that anytime there is block synchronization required in a CUDA implementation, the algorithm or the kernel is affected by the amount of memory transfer. Hence, a better synchronization mechanism is needed than the barrier synchronization technique implemented here. One possibility would be a signal network between the blocks or SMs dedicated just to synchronization.

Additional improvements can be achieved by the use of a better scheduling algorithm. In the current implementation, when a block is scheduled on an SM, according to the CUDA programming model, its execution has to completely finish before another block can be run in its place. Having the ability to switch a block on a SM after only a partial execution of that block can be advantageous. For example, a block that is sitting idle while waiting on another block to reach an execution point could be swapped out, allowing another process to execute instead. Obviously, if we try to use more blocks than the number of streaming multiprocessors in the current CUDA design, we could enter a deadlock situation, since an executing block could be waiting on the result from a block that has not been scheduled yet.

For the third step, computing the inverse of the triangular matrices, more control over the warp scheduler within the same SM could improve the performance by reducing the time spent synchronizing between threads of the same block.

For the first step, the *Ez* and *Zmn* matrix generation, there is little improvement expected since there is no data dependency between processors and the current implementation should be essentially optimum.

For problems with a memory footprint larger than the GPU can handle, the CPU carries a larger workload as it runs the same kernel over different iterations on the different parts of the problem. In that situation, multiple GPUs can be used to distribute the workload. All the steps of the algorithm except the LU factorization should see an improvement with multiple GPUs. For the LU factorization, the global synchronization

between blocks and the required data sharing are more difficult to implement with two physically separate GPU cards, so little improvement is expected.

REFERENCES

[1] A. F. Peterson, S. L. Ray and R. Mittra, *Computational Methods for Electromagnetics*, New York: IEEE Press, 1998.

[2] R. F. Harrington, *Field Computation by Moment Methods*, Malabar, FL: Krieger, 1982.

[3] R. F. Harrington, "Origin and development of the method of moments for field computation," in *Applications of the Method of Moments to Electromagnetic Fields*, ed. B. Strait, St. Cloud, FL: SCEEE Press, 1981.

[4] Canino, L. F., J.J. Ottusch, M.A. Stalzer, J.L Visher, & S.M. Wandzura, "Numerical solution of Helmholtz equation in 2D and 3D using a high-order Nyström descrization," *Journal of Computational Physics,* vol. 146, pp. 627-663, 1998.

[5] G. Liu, S.D. Gedney, "High-Order Nyström Solution of the Volume-EFIE for TM-Wave Scattering", *Microwave and Optical Technology Letters*, vol. 25, No. 1, pp. 8-11, April 5, 2000.

[6] S. Peng, Z. Nie. "Acceleration of the Method of Moments Calculations by Using Graphics Processing Units," *IEEE Trans. Antennas Propag.*, vol. 56, no. 7, pp. 2130-2133, 2008.

[7] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*, New York: Dover, 1965.

[8] A. F. Peterson and M. M. Bibby, *An Introduction to the Locally-Corrected Nyström Method*. *Synthesis Lectures on Computational Electromagnetics #25*. San Rafael: Morgan & Claypool, 2010.

[9] J. Strain, "Locally corrected multidimensional quadrature rules for singular functions," *SIAM J. Scientific Computing*, vol. 16, pp. 992-1017, 1995.

[10] "NVIDIA CUDA (Compute Unified Device Architecture): Programming Guide , Version 3.0." http://developer.download.nvidia.com/compute/cuda/3_0/docs/ NVIDIA_CUDA_Programming_Guide_3.0.pdf, Feb 2010.

[11] Dr. Dobb's, "CUDA, Supercomputing for Masses", <http://www.drdobbs.com/high-performance-computing/>

[12] D.K. Kola, V. Jalaparti, “Solving Linear System of Equations using Crout LU Decomposition”, IIT Kanpur, India.

[13] S. Xiao, W. Feng, “Inter-Block GPU Communication via Fast Barrier Synchronization”, Technical Report TR-09-19, Computer Science, Virginia Tech, 2009.